File Path: revideo-docs/api-reference/_category_.yml
label: API Reference
position: 10
link:
  type: generated-index

File Path: revideo-docs/api-reference/player-react/_category_.yml
label: '@revideo/player-react'
position: 1
link:
  type: generated-index

File Path: revideo-docs/api-reference/player-react/player.mdx
---
sidebar_position: 4
slug: /api/player-react/player

Player

You can embed your videos into your website using the Revideo react player.

Setup

Install the `@revideo/player-react` package into your project:

```bash
npm install @revideo/player-react
```

Usage

You have two options to use the player:

1. Through the CLI (recommended)

Inside your Revideo project directory, run:

```bash
npx revideo serve
```

This will start a local server that builds (and rebuilds) your project and
serves all required assets. You can then embed the player into your website like
this:

```tsx
import {Player} from '@revideo/player-react';
```

<Player src="http://localhost:4000/player" />;
```

### 2. Manually

If you don't want to use the CLI to serve your built project, you can build it manually by running:

```bash
npm run build
```

This will create an `out` directory in your project root. You can then copy all of these files to any public directory on your server and embed the player by specifying the path to the directory. Say you copied the files to `/public/outDir`, and the folder is accessible via `https://example.com/outDir`, you can embed the player like this:

```tsx
import {Player} from '@revideo/player-react';

<Player src="https://example.com/outDir" />;
```

Props

src

The path to your bundle folder.

controls (optional)

`boolean`

Disables the player controls when set to false. `default=true`

variables (optional)
```

`Record<string, any>`

Parameters / or variables passed to your video. See [here](/parameterized-video) learn more about parameterized videos. `default={}`

playing (optional)

`boolean`

Specifies if the player is currently playing. `default=false`

currentTime (optional)

`number`

Current time of the player in seconds.

looping (optional)

`boolean`

Controls whether the to loop back to the beginning when the player reaches the end of the video. `default=true`

width (optional)

`number`

Width of the underlying canvas element in pixels. It's recommended to leave this setting empty as it crops the scene instead of stretching it.
`default=undefined`

height (optional)

`number`

See `width`. `default=undefined`

quality (optional)

`number`

Reduces or increases the resolution of the canvas element without affecting the positioning of elements. `default=1`

onDurationChange (optional)

`(duration: number) => void`

Gets called every time the duration of the video changes (this can happen due to the input variables changing).

onTimeUpdate (optional)

`(time: number) => void`

Gets called every time the current time of the video is changed (every few ms during playback). Useful when visualizing the progress of the player outside of the player component.

File Path: revideo-docs/api-reference/renderer/_category_.yml
label: '@revideo/renderer'
position: 1
link:
  type: generated-index

File Path: revideo-docs/api-reference/renderer/renderPartialVideo.mdx
---
sidebar_position: 2
slug: /renderer/renderPartialVideo
---

renderPartialVideo()

The `renderPartialVideo()` function lets you render partial videos if you want to distribute the rendering workload across multiple workers. You can find an example of this in our
[Cloud Functions example](https://github.com/redotvideo/examples/tree/main/google-cloud-run-parallelized).
To use `renderPartialVideo()`, you don't have to manually assign a range of frames or timestamps to render. Instead, you just pass the worker id and the total number of workers your rendering job uses, and the function will figure out the frames to render by itself.

Since merging partial videos gives you audio issues (audio becomes laggy), this function returns the path to the audio file and mute video file of the partial video. Afterwards, you should first concatenate all of the partial audio files and then concatenate all of the partial mute video files, and then merge the full audio and video to obtain your final mp4 file.

To do this, you can use the `concatenateMedia()` ([docs](/ffmpeg/concatenateMedia)) and `mergeAudioWithVideo()` ([docs](/ffmpeg/mergeAudioWithVideo)) functions from `@revideo/ffmpeg`.

Example Usage

```tsx
import {renderPartialVideo} from '@revideo/renderer';

const {audioFile, videoFile} = renderPartialVideo({
  projectFile: './src/project.ts',
  variables: {color: 'white'},
  numWorkers: 10,
```

```
  workerId: 3,
  settings: {
    dimensions: [1080, 1792],
    logProgress: true,
    fmpeg: {
      ffmpegLogLevel: 'error',
      ffmpegPath: 'ffmpeg',
    },
    puppeteer: {
      args: ['--no-sandbox'],
    },
  },
});
```

Arguments

An object of type `RenderPartialVideoProps` with the following attributes:

projectFile:

string

A string pointing towards your Vite config file. This will probably be `./src/project.ts`.

workerId

number

The id of the worker. We start counting at 0, so if you have 5 workers, values from 0 to 4 are accepted.

numWorkers

number

The number of workers you use in total. This informs the function which range of the video to render. For instance, worker 0 out of 10 workers would render 1/10 of the full video, whereas worker 0 out of 2 would render half of the video.

variables?

Record<string, any>

Parameters / or variables passed to your video. See [here](/parameterized-video) learn more about parameterized videos.

settings?

A `Omit<RenderSettings, 'workers'>` object with the following properties:

outFile?

string, has to end with '.mp4'

The file name of the video output

outDir?

string

The output directory of the rendered video. `default="./output"`

range?

[number, number]

The start and end second of the video. Can be used to only render a part of the video.

dimensions?

[number, number]

Dimensions of the video to render as [x,y]. Uses the value specified in `project.meta` by default.

logProgress?

boolean

Logs render progress to the console if set to `true`.

ffmpeg?

FFmpeg options - is an instance of `FfmpegSettings`. These overwrite the following settings set through environment variables:

ffmpegLogLevel?

_`error` | `warning` | `info` | `verbose`, `debug` | `trace`_

The log level of FFmpeg. Can be one of `error`, `warning`, `info`, `verbose`, `debug`, `trace`. Default is `error`.

ffmpegPath?

The path to the FFmpeg binary. If not specified, the FFmpeg binary shipped with Revideo will be used.

puppeteer?

BrowserLaunchArgumentOptions

Launch options for puppeteer - is an instance of puppeteer's [BrowserLaunchArgumentOptions](https://pptr.dev/api/puppeteer.browserlaunchargumentoptions/)

viteBasePort?

number

The "base port" we use for the vite server. When you have three workers and a base port 5000, the vite servers created by the three workers will use port 5000, 5001, and 5002. Default is 9000.

viteConfig?

InlineConfig

Configuration of the vite server used for rendering, an instance of [InlineConfig](https://vitejs.dev/guide/api-javascript#inlineconfig). You can use these options to configure the server port, the cache directory, and more.

progressCallback?

(worker: number, progress: number) => void

A function that gets called with the progress of the rendering process, can be used to report progress back to users (e.g. in a web app). The function gets called with two arguments: the id of the worker that is calling the function, and the progress of the rendering process (float between 0 and 1). Does nothing by default

Return Value

{ audioFile: string, videoFile: string }

Paths to the audio and video files of the partial render.

File Path: revideo-docs/api-reference/renderer/renderVideo.mdx
---
sidebar_position: 1
slug: /api/renderer/renderVideo
---

renderVideo()

The `renderVideo` function lets you render (parameterized) videos in a nodejs process. It uses a headless browser to achieve this.

Example Usage

```tsx
import {renderVideo} from '@revideo/renderer';

function logProgressToConsole(id: number, progress: number) {
  console.log(`[${id}] Progress: ${(progress * 100).toFixed(1)}%`);
}

renderVideo({
  projectFile: './src/project.ts',
  variables: {color: 'white'},
  settings: {
    outFile: 'video.mp4',
    workers: 1,
    range: [1, 3],
    dimensions: [1080, 1792],
    logProgress: true,
    ffmpeg: {
      ffmpegLogLevel: 'error',
      ffmpegPath: 'ffmpeg',
    },
    puppeteer: {
      args: ['--no-sandbox'],
    }
    progressCallback: logProgressToConsole
```

```
  }
});
```

Arguments

The input arguments are a `RenderVideoProps` object, which has the following properties:

projectFile

string

Points towards your project file. This will probably be `./src/project.ts`.

variables?

Record<string, any>

Parameters / or variables passed to your video. See [here](/parameterized-video) learn more about parameterized videos.

settings?

A `RenderSettings` object with the following properties:

outFile?

string, has to end with '.mp4'

The file name of the video output

outDir?

string

The output directory of the rendered video. `default="./output"`

range?

[number, number]

The start and end second of the video. Can be used to only render a part of the video.

workers?

number

The number of processes you want to use to parallelize rendering, default is 1. Rendering a 100s long video with 10 workers means that 10 processes handle 10s of video each. Your laptop will probably render fastest with one worker, but VMs with a lot of computing power can benefit from using more than one

dimensions?

[number, number]

Dimensions of the video to render as [x,y]. Uses the value specified in `project.meta` by default.

logProgress?

boolean

Logs render progress to the console if set to `true`.

ffmpeg?

FFmpeg options - is an instance of `FfmpegSettings`. These overwrite the following settings set through environment variables:

ffmpegLogLevel?

_`error` | `warning` | `info` | `verbose`, `debug` | `trace`_

The log level of FFmpeg. Can be one of `error`, `warning`, `info`, `verbose`, `debug`, `trace`. Default is `error`.

ffmpegPath?

The path to the FFmpeg binary. If not specified, the FFmpeg binary shipped with Revideo will be used.

puppeteer?

BrowserLaunchArgumentOptions

Launch options for puppeteer - is an instance of puppeteer's [BrowserLaunchArgumentOptions](https://pptr.dev/api/puppeteer.browserlaunchargumentoptions/)

viteBasePort?

number

The "base port" we use for the vite server. When you have three workers and a base port 5000, the vite servers created by the three workers will use port 5000, 5001, and 5002. Default is 9000.

viteConfig?

InlineConfig

Configuration of the vite server used for rendering, an instance of [InlineConfig](https://vitejs.dev/guide/api-javascript#inlineconfig). You can use these options to configure the server port, the cache directory, and more.

progressCallback?

(worker: number, progress: number) => void

A function that gets called with the progress of the rendering process, can be used to report progress back to users (e.g. in a web app). The function gets called with two arguments: the id of the worker that is calling the function, and the progress of the rendering process (float between 0 and 1). Does nothing by default

Return Value

The function returns a string containing the path to the rendered video.

File Path: revideo-docs/api-reference/revideo-ffmpeg/_category_.yml
label: '@revideo/ffmpeg'
position: 2
link:
  type: generated-index

File Path: revideo-docs/api-reference/revideo-ffmpeg/concatenate-media.mdx
---
sidebar_position: 2
slug: /ffmpeg/concatenateMedia
---

concatenateMedia()

The `concatenateMedia()` function lets you concatenate video or audio files.

Example Usage

```tsx
import {concatenateMedia} from '@revideo/ffmpeg';

const audios = ['audio-0.wav', 'audio-1.wav', 'audio-2.wav'];
concatenateMedia(audios, 'audio.wav');
```

Arguments

files

List of strings containing paths to your input files.

outputFile

Path to where the output file will be saved.

File Path: revideo-docs/api-reference/revideo-ffmpeg/merge-audio-with-video.mdx
---
sidebar_position: 2
slug: /ffmpeg/mergeAudioWithVideo
---

mergeAudioWithVideo()

The `mergeAudioWithVideo()` function lets you merge audio with video files

Example Usage

```tsx
import {mergeAudioWithVideo} from '@revideo/ffmpeg';

mergeAudioWithVideo('audio.wav', 'visuals.mp4', 'out.mp4');
```

Arguments

audioPath

Path to the audio file

videoPath

Path to the video file

outputPath

Path to where the merged output file will be saved.

File Path: revideo-docs/code-snippets/_category_.yml
label: Code Snippets
position: 6
link:
  type: generated-index

File Path: revideo-docs/code-snippets/changing-object-size-over-time-with-signals.mdx
---
sidebar_position: 5
slug: /changing-object-size-over-time-with-signals

Changing Object Sizes with Signals

You can use signals to change values in your animation over time. Below you can find a example that uses signals to change the size of a circle over time.

```tsx editor
import {Circle, Txt, makeScene2D} from '@revideo/2d';
import {createSignal} from '@revideo/core';

export default makeScene2D(function* (view) {
  const circleSize = createSignal(50); // initial size of 50

  yield view.add(
    <>
      <Circle fill={'green'} size={circleSize} />
      <Txt fontSize={40} x={-300}>
        {() => `size: ${circleSize().toFixed(1)}`}
      </Txt>
    </>,
  );

  yield* circleSize(200, 2); // change size to 200 over two seconds
});
```

File Path: revideo-docs/code-snippets/hls-video.mdx
---
sidebar_position: 7
slug: /hls-video

HLS Video

Revideo supports the use of HLS video streaming, both for preview in the player and during rendering. Just note that you cannot use the fast webcodecs based video decoder for HLS video, and therefore, rendering will be slower than when using .mp4 files.

To use HLS video, simply set `src` to a file with an `.m3u8` ending in a `<Video/>` tag.

```tsx editor
import {Video, makeScene2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  yield view.add(
    <>
      <Video
        src={'https://test-streams.mux.dev/x36xhzz/x36xhzz.m3u8'}
        play={true}
        height={'100%'}
        time={5}
      />
    </>,
  );

  yield* waitFor(10);
});
```

File Path: revideo-docs/code-snippets/moving-objects.mdx
---
sidebar_position: 6
slug: /moving-manipulating-objects

Moving and Manipulating Objects

You can easily move objects and manipulate them using a variety of functions. Here, we move a square along with some text around the screen, make it larger, and finally turn it into a circle:

```tsx editor
import {Rect, Txt, makeScene2D} from '@revideo/2d';
import {all, waitFor, createRef, easeInBounce, easeInExpo} from '@revideo/core';

export default makeScene2D(function* (view) {
  const rectRef = createRef<Rect>();

  yield view.add(
    <Rect fill={'blue'} size={[100, 100]} ref={rectRef}>
      <Txt fontSize={30} fontFamily={'Sans-Serif'} fill={'white'}>
        Hi!
      </Txt>
    </Rect>,
  );

  yield* waitFor(0.5); // do nothing for 0.5s
  yield* all(rectRef().position.x(200, 1), rectRef().position.y(50, 1)); // move the rectangle to [200, 50] in 1s
  yield* all(rectRef().position.x(0, 2), rectRef().position.y(0, 2)); // move the rectangle to [0,0] (center) in 2s

  yield* rectRef().scale(2, 1); // scale the rectangle by 2 in 1s
  yield* rectRef().radius(100, 1); // increase the radius to 100 in 1s
  yield* waitFor(1); // do nothing for 1s
});
```

```
```

File Path: revideo-docs/code-snippets/streaming-text.mdx
---
sidebar_position: 1
slug: /streaming-text

Streaming Text

In many cases, you might want to stream texts to a video rather than having all of it appear at once.

One way to achieve this is by using the `.text()` method of the `<Txt>` tag.

```tsx editor
import {Txt, makeScene2D} from '@revideo/2d';
import {createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const textRef = createRef<Txt>();
  yield view.add(
    <Txt fontFamily={'Sans-Serif'} fill={'red'} ref={textRef}></Txt>,
  );

  yield* textRef().text('This is a text', 2);
});
```

The second argument (in this case, `2`) refers to the time it takes for the text to appear.

If you want more control over when words appear (for instance for captions with exact timestamps), you can repeatedly `.add()` text to your `<Txt>` node:

```tsx editor
import {Txt, makeScene2D} from '@revideo/2d';
import {createRef, waitFor} from '@revideo/core';
```

```
export default makeScene2D(function* (view) {
  const textRef = createRef<Txt>();
  yield view.add(
    <Txt fontFamily={'Sans-Serif'} fill={'red'} ref={textRef}></Txt>,
  );

  const words = ['This', 'is', 'a', 'text'];
  const secondsToAppear = [0.3, 0.6, 0.4, 0.2, 0.5];

  for (let i = 0; i < words.length; i++) {
    textRef().add(<Txt>{words[i]} </Txt>);
    yield* waitFor(secondsToAppear[i]);
  }
});
```

File Path: revideo-docs/code-snippets/transparent-video.mdx
---
sidebar_position: 6
slug: /transparent-video

Transparent Video

Revideo supports the use of transparent videos. This is useful when you want to overlay a video on top of other content, such as a background image or video.

Make sure the transparent video you're using is encoded using the `VP9` codec. These files usually have a `.webm` extension.

You can use command line tools such as `ffmpeg` to re-encode videos from one codec to another.

Note: There are other video codes that support transparency, such as `HEVC`, which might show up correctly in the preview through the ui or inside the player, which will lose their transparency when rendering.

```tsx editor
import {Img, Video, makeScene2D} from '@revideo/2d';
import {waitFor, createRef, useScene} from '@revideo/core';

export default makeScene2D(function* (view) {
  const avatarRef = createRef<Video>();
  const backgroundRef = createRef<Img>();

  yield view.add(
    <>
      <Img
        src={'https://revideo-example-assets.s3.amazonaws.com/mountains.jpg'}
        width={'100%'}
        ref={backgroundRef}
      />
      <Video
```

```
      src={'https://revideo-example-assets.s3.amazonaws.com/avatar.webm'}
      play={true}
      height={'100%'}
      ref={avatarRef}
    />
  </>,
 );

 yield* waitFor(avatarRef().getDuration());
});
```

File Path: revideo-docs/common-issues/_category_.yml
label: Common Issues
position: 7
link:
  type: generated-index

File Path: revideo-docs/common-issues/ffmpeg.mdx
---
sidebar_position: 1
slug: /common-issues/ffmpeg

FFmpeg Issues

Ffmpeg is a powerful tool for video editing. Revideo uses FFmpeg for multiple tasks, such as concatenating frames into a video, and audio manipulation among other things.

Revideo uses FFmpeg v6. To make installation easier, we ship a small version of FFmpeg with Revideo. It is automatically downloaded into the `node_modules` folder. This means that you do not need to install FFmpeg globally on your machine.

Below, we list some common issues with ffmpeg and how to fix them.

SIGSEGV (Segmentation Fault) when running Ffmpeg.

If you're on Linux, the ffmpeg version used in Revideo might cause a segmentation fault if you use remote urls for media files instead of local files.

You can normally fix this issue by installing nscd:

```
sudo apt-get install nscd
```

If you want to learn more about the root cause of this issue and its fix, check out
[this](https://stackoverflow.com/questions/60528501/ffmpeg-segmentation-fault-with-network-stream-source)
Stackoverflow thread,

Using a custom ffmpeg path

If you want to use a custom version of FFmpeg, you can either set the environment variable `FFMPEG_PATH` to the path of your FFmpeg binary, or you can
set the path to ffmpeg in the render function. Here is an example:

```tsx
import {renderVideo} from '@revideo/renderer';

// ...

await renderVideo('./src/project.ts', undefined, undefined, {
  ffmpeg: {
    // Points to the global ffmpeg installation,
    ffmpegPath: 'ffmpeg',
  },
});
```

See the [renderVideo](/api/renderer/renderVideo) API documentation for more information.

File Path: revideo-docs/common-issues/slow-rendering.mdx
---
sidebar_position: 2
slug: /common-issues/slow-rendering

Slow Rendering

The rendering speed of Revideo projects depends on the content of your video. Projects that contain no `<Video/>` elements generally render much faster than those that do, as processing videos is an expensive operation.

Here you can find some strategies that can help speed up rendering:

Upgrade to >=v0.4.4

We regularly release new versions of Revideo that often come with performance improvements. To get the best rendering speeds possible, make sure to upgrade Revideo to >=v0.4.4.

Use MP4 decoder

We use the Webcodecs API to extract frames from `<Video/>` elements during rendering. This, however, is currently only supported for mp4 files, and Revideo will use Ffmpeg-based decoder or naively seek set the `.currentTime` attribute on the HTMLVideoElement if it detects another file type. If Revideo fails to detect a file type, you will receive the following warning:

WARNING: Could not detect file type of video, will default to using mp4 decoder. If your video file is no mp4 file, this will lead to an error - to fix this, reencode your video as an mp4 file (better performance) or specify a different decoder.

If this occurs, you can set the decoder explicitly as a prop of your `<Video/>` tag:

```tsx
```

yield view.add(<Video src={'your_file'} decoder={'web'} />);
```

You should use the following decoder for different file types:

- `decoder={"web"}` for mp4
- `decoder={"ffmpeg"}` for .webm
- `decoder={"slow"}` for everything else

### Lower output resolution through `resolutionScale` in `project.meta`.

You can scale down the quality of your render output by setting the `resolutionScale` parameter in the `rendering` section of `project.meta`. We have noticed that setting the scale to 0.5 made our renders twice as fast as with the default of 1.0.

### Use smaller assets

When using media such as `<Img/>` or `<Video/>` elements, processing large files
takes Revideo longer than smaller ones. If you want to render a video with a resolution of 1920x1080, it would be overkill to insert a 4K video inside a `<Video/>` tag, as the resolution will be scaled down anyway. In this case, you can speed up renders by using a smaller file.

### Parallelize Rendering

You can parallelize your rendering workloads using the `workers`: argument in `renderVideo()` ([docs](https://docs.re.video/api/renderer/renderVideo/)). Note that this can get memory-intensive, as it spins up multiple ffmpeg workers in parallel - you might have to switch to more powerful hardware to support a lot of workers.

Alternatively, you can also parallelize render workloads using `renderPartialVideo()`
([docs](https://docs.re.video/renderer/renderPartialVideo/)) and serverless function services such as Google Cloud Run or AWS Lambda. We have released an
```

[example project](https://github.com/redotvideo/examples/tree/main/google-cloud-run-parallelized)
demostrating how you can set up parallel rendering with Google Cloud Functions, and will release a guide for AWS Lambda soon.

We will soon release a managed platform that enables fast, parallelized rendering. If you are interested in getting access, feel free to register interest [here](https://tally.so/r/mOz4GK).

File Path: revideo-docs/get-help/_category_.yml
label: Get Help
position: 11
link:
  type: generated-index

File Path: revideo-docs/get-help/discord.mdx
Discord

You can join our Discord community where the Revideo team actively responds to issues. [Here](https://discord.gg/MVJsrqjy3j) is an invite link.

How to ask questions

We can best help you with your questions if you give us as much information as possible. When you run into an issue with Revideo and want to ask a question, please do the following:

Investigate yourself

- Can you isolate which piece of your code causes the issue? Try deleting the most recent lines of code that caused the issue
- If you are having problems when using a video / audio file: Try changing the file to see if the error disappears
- If you are deploying Revideo in the cloud and are facing an issue: check if everything works locally on your laptop

Provide Context

When asking a question, please provide the following information in your question:

- Which version of Revideo are you running? You can find this in your `package.json`
- Have you found anything noteworthy while investigating the issue yourself (see above on what to investigate)?
- Are you using a Revideo template? If so, which one? What have you modified?
- Please describe what triggers the issue. Does it occur when you render a video, when you start the editor, or when you click a certain button?
- Please provide the full error logs
- If possible, please share (relevant parts of) your code with us.

We understand that this might take some extra effort, but we promise that this

will help us get to a solution for your issue faster!

When messaging us on the Discord channel, please include these bullet points in your message. Thank you!

File Path: revideo-docs/guide/_category_.yml
label: Usage Guide
position: 2
collapsed: false
link:
  type: generated-index

File Path: revideo-docs/guide/building-webapps/_category_.yml
label: Building Web Apps
position: 100
link:
  type: generated-index

File Path: revideo-docs/guide/building-webapps/deploy-rendering-service.mdx
---
sidebar_position: 11
slug: /rendering-in-production

Deploying a Rendering Service in Production

Rendering videos is a relatively expensive operation compared to most other requests served in a web app - you should therefore pay attention to which hardware and deployment setup you want to use to deploy a rendering service for your Revideo apps.

You should expect a rendering job (a function call to `renderVideo()` or `renderPartialVideo()` ([API Reference](/api/renderer/renderVideo))) to **require at least 8-10GB of RAM** to run fast and without issues. When multiple rendering jobs run at the same time, they will get slower if more RAM is not available.

Below, we discuss a few possibilities of how you can deploy a rendering service in production:

Parallelized Rendering with Serverless Functions (Recommended)

To speed up rendering in production, it's useful to parallelize rendering with serverless functions using `renderPartialVideo()` ([docs](https://docs.re.video/renderer/renderPartialVideo)). When rendering a 10 minute video, parallelizing across 10 workers would spin up 10 serverless functions that each render one minute of the video. We have created example guides for parallelized rendering on AWS Lambda and Google Cloud Functions.

Note: In our experiments, parallelized Rendering on AWS Lambda performs significantly faster due to much better cold start times. We highly recommend using AWS Lambda for rendering instead of Google Cloud Functions

Parallelized Rendering on AWS Lambda

You can find the example project for parallelized rendering on AWS Lambda along with a setup guide [here](https://github.com/redotvideo/examples/tree/main/parallelized-aws-lambda)

Parallelized Rendering with Google Cloud Functions

You can find the example project for parallelized rendering with Cloud Functions along with a setup guide [here](https://github.com/redotvideo/examples/tree/main/google-cloud-run-parallelized)

Single-process Rendering

If your videos are short and / or rendering speeds are not a huge factor for you, you can render videos across a single process instead of setting up parallelized rendering. For example, here is a super simple express server for rendering a video:

```ts
import {renderVideo} from '@revideo/renderer';
import {v4 as uuidv4} from 'uuid';
import * as express from 'express';
import * as fs from 'fs';
import * as path from 'path';

const app = express();
app.use(express.json());

app.get('/', (req, res) => {
  res.status(200).send(`Hello World!`);
});

app.post('/render', async (req, res) => {
  try {
    const {variables} = req.body;
    const jobId = uuidv4();
```

```
    console.log('Rendering video...');
    await renderVideo({
      projectFile: './src/project.ts',
      variables,
      settings: {outFile: `${jobId}.mp4`, logProgress: true},
    });
    console.log('Finished rendering');

    const outputFilePath = path.join(process.cwd(), `./output/${jobId}.mp4`);

    if (fs.existsSync(outputFilePath)) {
      res.sendFile(outputFilePath); // alternatively (and recommended), upload file to a
bucket
    } else {
      res.status(500).send('Rendered video not found');
    }
  } catch (err) {
    console.error('Error rendering video:', err);
    res.status(500).send('Error rendering video');
  }
});

const port = parseInt(process.env.PORT) || 8000;
app.listen(port, () => {
  console.log(`listening on port ${port}`);
});
```

You can deploy a server like this on a normal VM instance or on a serverless deployment platform like Google Cloud Run.

Revideo platform

We are building a cloud platform that makes it easy to deploy Revideo projects and uses infrastructure optimized for fast rendering speeds. You can sign up to its waitlist [here](https://tally.so/r/mOz4GK).

File Path: revideo-docs/guide/building-webapps/rendering-endpoint.mdx
---
sidebar_position: 2
slug: /render-endpoint

Local Development with the CLI

To build web apps on your laptop, we recommend using our CLI to deploy your revideo project, which will expose a web server with endpoints for rendering videos and downloading rendered videos as mp4. An application example of this can be found in our
[Saas template](https://github.com/redotvideo/examples/saas-template).

To deploy a Revideo project, run the following command:

```bash
npx revideo serve --projectFile ./src/project.ts --port 3000
```

The `--port` parameter is optional. By default, the service will use port 4000.

Starting a render job

Once your Revideo project is deployed, you can render videos using the `/render` endpoint by passing your desired `settings` and `variables` as a parameter. This endpoint will trigger a render process on the server, and respond with a download url once the rendering process has finished.

Example request:

```bash
curl -X POST http://your-revideo-service.com/render \
  -H "Content-Type: application/json" \
  -d '{
    "variables": {
      "image": "some-image.png",
```

```
    "color": "red"
  },
  "settings": { "workers": 2 }
 }'
```

If you want to report the rendering progress back to the client, you can set the `streamProgress` parameter to true:

```bash
curl -X POST http://your-revideo-service.com/render \
 -H "Content-Type: application/json" \
 -d '{
  "variables": {
    "image": "some-image.png",
    "color": "red"
  },
  "settings": { "workers": 2 },
  "streamProgress": true
 }'
```

Rendering With Callbacks

If you don't want to keep the connection to the server open during the full duration of the rendering process, you can render videos with the `callbackUrl` parameter. This will trigger a render process on the server, and respond to a callback url after the rendering process is done.

Example request:

```bash
curl -X POST http://your-revideo-service.com/render \
 -H "Content-Type: application/json" \
 -d '{
  "variables": {
    "image": "some-image.png",
```

```
    "color": "red"
  },
  "callbackUrl": "http://your-callback-url.com/render-status"
}'
```

In this request, `variables` refer to the variables passed to your video, and `callbackUrl` is the url that will receive an update when the rendering process has finished. The immediate response to the request will contain a `jobId` that lets the client distinguish between status updates from different rendering jobs.

Example response:

```json
{
  "jobId": "123e4567-e89b-12d3-a456-426614174000"
}
```

Downloading rendered videos

Once the render job is finished or has failed, the revideo servide will send an update to the specified callback url via a POST request. The response will contain a download link from which you can download the exported video.

Example Callback Response (success):

```bash
curl -X POST <callbackUrl> \
-H "Content-Type: application/json" \
-d '{
  "jobId": "123e4567-e89b-12d3-a456-426614174000",
  "status": "success",
  "downloadLink": "http://your-revideo-service.com/download/42078492-fbb9-4570-a329-785e87456618.mp4"
}'
```

```

You can now download the file using the download link:

```
GET http://your-revideo-service.com/download/42078492-fbb9-4570-a329-785e87456618.mp4
```

**Note:** You should not use this link as a method of permanently serving files (for example to provide a download link to your users). Instead, it should only be used to download the file once, and afterwards the file should be stored on a permanent storage solution like a bucket. By default, the file will be deleted 10 minutes after the success callback.

**Example Callback Response (error):**

```bash
curl -X POST <callbackUrl> \
-H "Content-Type: application/json" \
-d '{
  "jobId": "123e4567-e89b-12d3-a456-426614174000",
  "status": "error",
  "error": "<error message>"
}'
```
```

File Path: revideo-docs/guide/building-webapps/saas-template.mdx
---
sidebar_position: 12
slug: /saas-template

Revideo SaaS Template

The Revideo SaaS Template is a minimal NextJS app that lets users preview and make changes to a video, and also lets them export their video to mp4.

This project is the best starting point if you want to build a web app with Revideo. You can find its code in the corresponding
[Github repository](https://github.com/redotvideo/examples/tree/main/saas-template)
or select it as a template when running `npm init @revideo@latest`.

File Path: revideo-docs/guide/building-webapps/using-the-player.mdx
---
sidebar_position: 1
slug: /preview-with-player

Video Preview with Player

Revideo provides a React Player component
([API reference](/api/player-react/player)) to embed Revideo projects into your React or NextJS web app. The component lets you preview videos and changes made
to your variables in real-time without forcing you to export your project to mp4 beforehand. A full example of this can be found in our
[SaaS Template](https://github.com/redotvideo/examples/tree/main/saas-template).

To display a project using the player, you first need to build your Revideo project, give your web app access to the built project file and reference it through the `src` prop of the `<Player/>`. If you use custom css (e.g. to specify fonts) or local assets (through the `/public` folder in your Revideo project), you will also have to make these assets available to your web app (either through serving them or copying over files).

Revideo provides a CLI command to serve your project that automatically serves your Revideo project and rebuilds it when you make changes, which you can use during development along with the Player.

Using the Player with the CLI (recommended)

Inside your Revideo project, run:

```bash
npx revideo serve
```

This will start to serve the project located in `./src/project.ts` on port 4000

(you can change these values using the `--port` and `--projectFile` flags). Now, inside your web app, you can use the player like this:

```tsx
import {Player} from '@revideo/player-react';

<Player src="http://localhost:4000/player/" />;
```

The `serve` command in the CLI is the recommended and most convenient option for
local development. It automatically builds your project and serves all the required assets (such as your project file, assets in `/public`, and css), so that you don't have to copy them over into your web app - what you see inside the player in your web app should be the same as what you see in the editor that you see when you run `npm start`. When you make changes to your Revideo project,
you also won't have to rebuild your project, as the CLI watches for changes and will handle them automatically.

Using the Player manually

If you don't want to use the CLI to serve your built project, you can build it manually by running:

```bash
npm run build
```

This will create a directory in your project root containing your built project, typically in `dist`. You can then copy all of these files to any public
directory on your web app server and embed the player by specifying the path to the directory. Say you copied the files to `/public/outDir` inside your web app, and the folder is accessible via `https://example.com/outDir` - you can then embed the player like this:

```tsx
```

```
import {Player} from '@revideo/player-react';

<Player src="https://example.com/outDir/" />;
```

File Path: revideo-docs/guide/designing-animations/3d-animations-with-threejs.mdx
---
sidebar_position: 6
slug: /3d-animations-with-threejs

3D Animations with Three.js

[Three.js](https://threejs.org/) is an animation library that can be used to build 3d animations. Since it can render to an HTML canvas, it can be used together with Revideo and make use of revideo-specific features such as [tweening](/tweening) and [signals](/signals).

A project with a custom `<Three/>` component can be found on [Github](https://github.com/redotvideo/examples/tree/main/three-js-example) and
serves as a great starting point to build 3d animations:

```tsx
import {Three} from '../components/Three';
import {makeScene2D, Txt} from '@revideo/2d';
import {
  tween,
  waitFor,
  delay,
  createRef,
  all,
  chain,
  linear,
} from '@revideo/core';
import * as THREE from 'three';

function setup3DScene() {
  const threeScene = new THREE.Scene();

  const geometry = new THREE.BoxGeometry(0.2, 0.2, 0.2);
```

```
  const material = new THREE.MeshNormalMaterial();

  const mesh = new THREE.Mesh(geometry, material);
  threeScene.add(mesh);

  const camera = new THREE.PerspectiveCamera(90);

  mesh.position.set(0, 0, 0);
  mesh.scale.set(1, 1, 1);
  camera.rotation.set(0, 0, 0);
  camera.position.set(0, 0, 0.5);

  return {threeScene, camera, mesh};
}

export default makeScene2D(function* (view) {
  const {threeScene, camera, mesh} = setup3DScene();

  const threeRef = createRef<Three>();
  const txtRef = createRef<Txt>();

  yield view.add(
    <>
      <Three
        width={1920}
        height={1080}
        camera={camera}
        scene={threeScene}
        opacity={0}
        fontWeight={900}
        ref={threeRef}
      />
    </>,
  );

  yield view.add(
    <Txt fill={'black'} fontFamily={'Lexend'} ref={txtRef} fontSize={80} />,
```

```
  );

  yield* chain(
    txtRef().text('Revideo x 3D with Three.js', 1),
    all(txtRef().position.y(-300, 1), delay(0.5, threeRef().opacity(1, 0.5))),
  );

  yield tween(4, value => {
    mesh.rotation.set(0, linear(value, 0, 2 * 3.14), 0);
  });

  yield* waitFor(2);

  yield addRotatingCube(threeRef().scene(), 0.1, 0.3, -0.2, 0.1);
  yield addRotatingCube(threeRef().scene(), 0.1, -0.3, -0.2, 0.1);

  yield* waitFor(2);
});

function* addRotatingCube(
  threeScene: THREE.Scene,
  size: number,
  x: number,
  y: number,
  z: number,
) {
  const geometry = new THREE.BoxGeometry(size, size, size);
  const material = new THREE.MeshNormalMaterial();
  const mesh = new THREE.Mesh(geometry, material);

  mesh.position.set(x, y, z);
  mesh.scale.set(1, 1, 1);

  threeScene.add(mesh);

  yield* tween(4, value => {
    mesh.rotation.set(0, linear(value, 0, 2 * 3.14), 0);
```

```
  });
}
```

File Path: revideo-docs/guide/designing-animations/_category_.yml

```
label: Designing Videos
position: 99
link:
  type: generated-index
```

File Path: revideo-docs/guide/designing-animations/designing-animations.mdx
---
sidebar_position: 1
slug: /designing-animations

Motion Canvas

Revideo was forked from [Motion Canvas](https://motioncanvas.io/) - therefore, both projects largely share the same API for designing animations. If you want to learn how to build animations, you should read through the [Motion Canvas Guide](/category/motion-canvas-guide). API changes that are specific to Revideo (such as the addition of [Rive animations](/rive-animations)) or relevant for Revideo's use cases, are listed in the following pages.

If you like to learn from examples, you can check out our [Code Snippets](/category/code-snippets), where you can find code examples of common animations, such as [streaming text](/streaming-text) or [moving objects](/moving-manipulating-objects).

File Path: revideo-docs/guide/designing-animations/emojis.mdx
---
sidebar_position: 4
slug: /emojis

Emojis

Using Emojis can cause issues as they might be rendered inconsistently across different browser versions. To get consistent results for rendering emojis, you should explicitly use a font that supports a variant of emojis you like. You can read in detail about how to use fonts [here](/custom-font).

An example of a font that (only) supports emojis is `Noto Color Emoji`. If your main font for text is `Lexend` and you want to use `Noto Color Emoji` for emojis, you can use the following css in `src/global.css`:

```css
@import url('https://fonts.googleapis.com/css2?
family=Lexend:wght@600&family=Noto+Color+Emoji&display=swap');
```

Now, you can import the css file in `src/project.ts` and select the fonts for your `<Txt/>` nodes:

src/project.ts:

```tsx
import {makeProject} from '@revideo/core';
import example from './scenes/example?scene';
import './global.css';

export default makeProject({
  scenes: [text],
});
```

src/scenes/example.tsx:

```tsx
import {Txt, makeScene2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  yield view.add(
    <Txt text={'Hello ☒     '} fontFamily={"Lexend, 'Noto Color Emoji'"} />,
  );

  yield* waitFor(1);
});
```

File Path: revideo-docs/guide/designing-animations/logical-separation.mdx
---
sidebar_position: 3
slug: /logical-separation

Logical Separation of Scene Components

A question that frequently comes up from Revideo users is how to organize your scene code to achieve good logical separation and make things readable.

Custom Generator Functions

Revideo lets you define your scene code as a generator function wrapped in `makeScene2D`. Here is a mininal example that displays an image for five seconds:

```tsx
import {Img, makeScene2D, View2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
 yield view.add(
  <Img
   width={'30%'}
   ref={logoRef}
   src={
    'https://revideo-example-assets.s3.amazonaws.com/revideo-logo-white.png'
   }
  />,
 );

 yield* waitFor(5);
});
```

Let's say that you additionally want to display some text:

```tsx
import {Img, Txt, makeScene2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  yield view.add(
    <Img
      width={'30%'}
      ref={logoRef}
      src={
        'https://revideo-example-assets.s3.amazonaws.com/revideo-logo-white.png'
      }
    />,
  );

  yield view.add(<Txt fill="red" y={300} text={'Hello World!'} />);

  yield* waitFor(5);
});
```

Instead of adding the text node inside your main function, you can also create a separate generator function that does so and is called within your main generator function:

```tsx
import {Img, Txt, makeScene2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  yield view.add(
    <Img
      width={'30%'}
      ref={logoRef}
      src={
        'https://revideo-example-assets.s3.amazonaws.com/revideo-logo-white.png'
```

```
    }
   />,
  );

  yield addText(view, 'Hello World!');

  yield* waitFor(5);
});

function* addText(view: View2D, displayText: string) {
  yield view.add(<Txt fill="red" y={300} text={displayText} />);

  yield* waitFor(5);
}
```

Especially as your scenes get more complex, it makes sense to separate different parts of them (for example, you might have your main video definition in one function and subtitling logic in another one).

Calling generator functions with `yield` and `yield*`:

When you have multiple generator functions that you want to use, it is important to mention the difference between `yield` and `yield*`:

`yield` will call your generator function but not wait for it to finish executing before executing the rest of your code. If you have a generator function `displaySubtitles()`, this will make a big difference:

Say you have the following code:

```tsx
yield displaySubtitles();
// rest of your scene code, will get displayed at the same time as subtitles
```

This will display your subtitles along the rest of your scene code - it will not

wait for all subtitles to have finished.

On the other hand, if you use `yield*`, the main generator function will wait for `displaySubtitles()` to finish executing before displaying your remaining animations:

```tsx
yield * displaySubtitles(); // wait for displaySubtitles to finish
// rest of your scene code, will get displayed after subtitles
```

If you have two generator functions that you want to display at the same time, you can therefore call both subsequently with `yield`:

```tsx
yield displaySubtitles();
yield displayImages();
```

Alternatively, you can use the `all` function:

```tsx
yield * all(displaySubtitles(), displayImages());
```

If you want to display functions subsequently, you can call them subsequently with `yield*` or use `chain`:

```tsx
yield * displaySubtitles();
yield * displayImages();
```

This yields the same result as:

```tsx
yield * chain(displaySubtitles(), displayImages());
```

```

You can learn more about controlling the animation flow with functions like
`all` and `chain` [here](/flow).

## Custom Components

You can build your own `<CustomComponent/>` to use in your revideo projects. A
good guide for this is available in the
[Motion Canvas guide](/custom-components).

**Note:** In most cases, you won't have to implement a custom `draw()` function
when building a custom component. If you do, note that `draw()` functions in
Revideo need to be implemented as async functions, while they are synchronous
in
Motion Canvas.
```

File Path: revideo-docs/guide/designing-animations/rive-animations.mdx
---
sidebar_position: 5
slug: /rive-animations

Rive Animations

You can use our `<Rive/>` component to integrate animations built with [Rive](https://rive.app) inside your videos:

```tsx
import {Rive, makeScene2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  yield view.add(
    <Rive
      src={'https://revideo-example-assets.s3.amazonaws.com/emoji.riv'}
      animationId={1}
      size={[600, 600]}
    />,
  );

  yield* waitFor(5);
});
```

The `<Rive/>` component is a child of `<Rect/>`. Additionally, it has the following props:

src

string

Points towards your `.riv` file

animationId?

string | _number_

The identifier of your animation, either as a string or an index. Will use default animation if not selected

artboardId?

string | _number_

The identifier of your artboard, either as a string or an index. Will use default artboard if not selected

File Path: revideo-docs/guide/installation-and-setup.mdx
---
sidebar_position: 1
slug: /installation-and-setup
---

import AnimationPlayer from '@site/src/components/AnimationPlayer';
import CodeBlock from '@theme/CodeBlock';
import source from '!!raw-loader!@revideo/examples/src/scenes/quickstart';

Quickstart

Let's install revideo with a sample project. To use revideo, make sure that you have [Node.js](https://nodejs.org/) version 16 or greater.

:::info

If you're on Linux, also make sure that you have nscd installed:
`sudo apt-get install nscd`. You need this package for
[ffmpeg](/common-issues/ffmpeg).

:::

Creating a new project

Run the following command to create a new revideo project (If this fails, check out the [troubleshooting](#troubleshooting) section):

```bash
npm init @revideo@latest
```

Now, select the **default project**, navigate to its folder and install all dependencies:

```shell
cd <project-path>
```

npm install
```

To preview your video in the editor, run:

```shell
 npm start
```

The editor can now be accessed by visiting
http://localhost:9000/. Here you should see the video
shown below.

<video width="708" height="400" controls>
 <source
  src="https://revideo-example-assets.s3.amazonaws.com/revideo-example.mp4"
  type="video/mp4"
 />
 Your browser does not support the video tag.
</video>

```tsx
import {Audio, Img, Video, makeScene2D} from '@revideo/2d';
import {all, chain, createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
 const logoRef = createRef<Img>();

 yield view.add(
  <>
   <Video
    src={'https://revideo-example-assets.s3.amazonaws.com/stars.mp4'}
    play={true}
    size={['100%', '100%']}
   />
   <Audio
    src={'https://revideo-example-assets.s3.amazonaws.com/chill-beat.mp3'}

```
      play={true}
      time={17.0}
    />
  </>,
 );

 yield* waitFor(1);

 view.add(
  <Img
   width={'1%'}
   ref={logoRef}
   src={
    'https://revideo-example-assets.s3.amazonaws.com/revideo-logo-white.png'
   }
  />,
 );

 yield* chain(
  all(logoRef().scale(40, 2), logoRef().rotation(360, 2)),
  logoRef().scale(60, 1),
 );
});
```

Rendering the Video

You can render your video by running:

```
npm run render
```

This will call the `./src/render.ts` script in your code:

```ts
import {renderVideo} from '@revideo/renderer';
```

```
async function render() {
  console.log('Rendering video...');

  // This is the main function that renders the video
  const file = await renderVideo({
    projectFile: './src/project.ts',
    settings: {logProgress: true},
  });

  console.log(`Rendered video to ${file}`);
}

render();
```

For more information, check out the [renderVideo()](/api/renderer/renderVideo) API reference.

Alternatively, you can also render your video using the button in the editor that starts when you run `npm run start`.

Rendering from the browser

To render videos from the editor, simply click the "Render" Button:

![Render Button](https://revideo-example-assets.s3.amazonaws.com/render-button.png)

Understanding the Video Code

For now, we can ignore all files in our revideo project except for `src/scenes/example.tsx`, as this is where the visuals and audio of our video are defined. Let's walk through all the parts of the code that might confuse you, and provide explanations and references.

Defining a generator function

Our animation is defined within a generator function that is passed to `makeScene2D` - this function describes the sequence of events happening in our video:

```tsx
import {Audio, Img, Video, makeScene2D} from '@revideo/2d';
import {all, chain, createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
// your animation flow here
}
```

Generator functions can return multiple values - when they are called, they will execute until they first encounter a `yield` statement, and return the yielded value. Revideo renders animations by continually calling the generator function, which will yield frames that we can export. It is not neccessary to understand how this works exactly, but if you are interested, you can read about the animation flow in revideo [here](/flow).

Adding Video and Audio elements

At the start of our generator function, we add [Video](/api/2d/components/Video) and [Audio](/api/2d/components/Audio) tags to our `view`, which will display them on the canvas. Other components you could add include [Txt](/api/2d/components/Txt) or [Img](/api/2d/components/Img) tags, or basic shapes like [Rect](api/2d/components/Rect) or [Circle](api/2d/components/Circle). You can find the API for all components [here](api/2d/components).

```tsx
yield view.add(
 <>
  <Video
   src={'https://revideo-example-assets.s3.amazonaws.com/stars.mp4'}
   play={true}
```

```
    size={['100%', '100%']}
   />
   <Audio
    src={'https://revideo-example-assets.s3.amazonaws.com/chill-beat.mp3'}
    play={true}
    time={17.0}
   />
  </>,
);
```

A few points about input arguments:

- In both cases, `src` refers to the file, which points to a remote url on the bucket. Alternatively, you can use local files by passing their path.
- Passing `size={["100%", "100%"]}` makes the video stretch to the full width and height of its canvas.
- Adding `play={true}` makes both media elements play immediately, instead of being in a paused state.

Play media for one second

After adding our background video and audio, we execute

```tsx
yield * waitFor(1);
```

The function [waitFor](/api/core/flow/#waitFor) does - as the name suggests - nothing. It is particularly useful when waiting for media (like videos and audio) to play or when we want to have a still-standing image.

Animating the revideo logo

Lastly, we let the revideo logo spin into our video:

```tsx
```

```
view.add(
 <Img
  width={'1%'}
  ref={logoRef}
  src={
   'https://revideo-example-assets.s3.amazonaws.com/revideo-logo-white.png'
  }
 />,
);

yield *
 chain(
  all(logoRef().scale(40, 2), logoRef().rotation(360, 2)),
  logoRef().scale(60, 1),
 );
```

A few things happen here: First, we add the revideo logo as an [Img](/api/2d/components/Img) to our canvas. We set its initial width to only 1% of the screen, as we want it to grow as the video plays. We also pass a [reference](/references) through `ref={logoRef}`, which we had initialized before. Like [React refs](https://react.dev/learn/referencing-values-with-refs), references allow us to access and modify the behavior of elements after they've been initialized.

In our code, we use a reference to the revideo logo to later animate it. Particularly, we run the following commands:

- `scale(x, s)`: scales the size of the logo to `x` times its original size, within `n` seconds.
- `rotation(d, s)`: rotates the image by `d` degrees within `s` seconds

The flow of these animations is determined by the keywords [chain](/flow/#chain) and [all](/flow/#all). The former instructs revideo to play its input animations after one another, while the latter instructs revideo to play them simultaneously. As a result, we first see the revideo logo rotating around 360 degrees while growing to 40x its original size. After this is done, the logo

still grows to 60x its original size. You can learn more about the animation flow in revideo [here](/flow).

<!---

Now save the file. Any changes you make are automatically picked up and reflected in the preview.

You should see a red circle in the preview pane at the top right of the web application. Press the play button to see the circle animate across the screen.

Explanation

Each video in Motion Canvas is represented by a project configuration object. In our example, this configuration is declared in `src/project.ts`:

```ts title="src/project.ts"
import {makeProject} from '@revideo/core';

import example from './scenes/example?scene';

export default makeProject({
  scenes: [example],
});
```

When creating a project, we need to provide it with an array of scenes to display. In this case, we use only one scene imported from `src/scenes/example.tsx`.

A scene is a set of elements displayed on the screen and an animation that governs them. The most basic scene looks as follows:

```tsx
import {makeScene2D} from '@revideo/2d';

export default makeScene2D(function* (view) {
```

```
  // animation
});
```

`makeScene2D()` takes a function generator and turns it into a scene which we then import in our project file. The function generator describes the flow of the animation, while the provided `view` argument is used to add elements to the scene.

You can learn more about scenes, nodes, and this XML-like syntax in the [scene hierarchy](/hierarchy) section. For now, what's important is that, in our example, we add an individual [`<Circle/>`](/api/2d/components/Circle) node to our scene. We make it red, set its width and height as `140` pixels and position it `300` pixels left from the center:

```tsx
view.add(
  <Circle
    // highlight-start
    ref={myCircle}
    x={-300}
    width={140}
    height={140}
    fill="#e13238"
  />,
);
```

To animate our circle we first need to [grab a reference to it](/references). That's the purpose of the [`createRef`](/api/core/utils#createRef) function. We use it to create a reference and pass it to our circle using the [`ref`](/api/2d/components/NodeProps#ref) attribute:

```tsx
// highlight-next-line
const myCircle = createRef<Circle>();
```

```
view.add(
  <Circle
    // highlight-next-line
    ref={myCircle}
    x={-300}
    width={140}
    height={140}
    fill="#e13238"
  />,
);
```

We then access the circle through `myCircle()` and animate its properties:

```tsx
yield *
  all(
    myCircle().fill('#e6a700', 1).to('#e13238', 1),
    myCircle().position.x(300, 1).to(-300, 1),
  );
```

This snippet may seem a bit confusing so let's break it down.

Each property of a node can be read and updated throughout the animation. For example, in the circle above we defined its `fill` property as `'#e13238'`:

```tsx
<Circle
  ref={myCircle}
  x={-300}
  width={140}
  height={140}
  // highlight-next-line
  fill="#e13238"
/>
```

```

Using our reference we can now retrieve this property's value:

```ts
const fill = myCircle().fill(); // '#e13238'
```

We can also update it by passing the new value as the first argument:

```ts
myCircle().fill('#e6a700');
```

This will immediately update the color of our circle. If we want to transition to a new value over some time, we can pass the transition duration (in seconds) as the second argument:

```ts
myCircle().fill('#e6a700', 1);
```

This [creates a tween animation](/tweening) that smoothly changes the fill color over one second.

Animations in Motion Canvas don't play on their own, we need to explicitly tell them to. This is why scenes are declared using generator functions - they serve as a description of how the animation should play out. By yielding different instructions we can tell the scene animation to do different things.

For example, to play the tween we created, we can do:

```ts
yield * myCircle().fill('#e6a700', 1);
```

This will pause the generator, play out the animation we yielded, and then
```

continue.

To play another animation, right after the first one, we can simply write another `yield*` statement:

```ts
yield * myCircle().fill('#e6a700', 1);
yield * myCircle().fill('#e13238', 1);
```

But since we're animating the same property, we can write it in a more compact way:

```ts
yield * myCircle().fill('#e6a700', 1).to('#e13238', 1);
```

In our example, aside from changing the color, we also move our circle around. We can try doing it the same way we animated the color:

```ts
yield * myCircle().fill('#e6a700', 1).to('#e13238', 1);
yield * myCircle().position.x(300, 1).to(-300, 1);
```

This works, but the position will start animating **after** the fill color. To make them happen at the same time, we use the [`all`][all] function:

```ts
yield *
  all(
    myCircle().fill('#e6a700', 1).to('#e13238', 1),
    myCircle().position.x(300, 1).to(-300, 1),
  );
```

[`all`][all] takes one or more animations and merges them together. Now they'll

happen at the same time. The [animation flow](/flow) section goes into more depth about generators and flow functions such as [`all`][all].

This brings us back to our initial example:

<CodeBlock language="tsx" title="src/scenes/example.tsx">
 {source}
</CodeBlock>

---->

Troubleshooting

<details>
 <summary>
  <code>npm init @revideo@latest</code> fails to execute.
 </summary>

There was [a bug in npm](https://github.com/npm/cli/issues/5175) causing the above command to fail. It got fixed in version `8.15.1`. You can follow [this guide](https://docs.npmjs.com/try-the-latest-stable-version-of-npm) to update your npm. Alternatively, you can use the following command instead:

```bash
npm exec @revideo/create@latest
```

</details>
<details>
 <summary>
  <code>npm install</code> fails with <code>code ENOENT</code>
 </summary>

If `npm install` fails with the following error:

```bash
npm ERR! code ENOENT
```

npm ERR! syscall open
npm ERR! path [path]\package.json
npm ERR! errno -4058
npm ERR! enoent ENOENT: no such file or directory, open '[path]\package.json'
npm ERR! enoent This is related to npm not being able to find a file.
npm ERR! enoent
```

Make sure that you're executing the command in the correct directory. When you finish bootstrapping the project with `npm init`, it will display three commands:

```bash
cd [path]
npm install
npm start
```

Did you run the `cd` command to switch to the directory containing your project?

</details>
<details>
 <summary>
  I moved the camera too far away and can't find the preview (The preview is is black)
 </summary>

You can press `0` to refocus the camera on the preview.

</details>
<details>
 <summary>
  The animation ends abruptly or does not start at the beginning.
 </summary>

Make sure the playback range selector in the timeline starts and ends where you expect it to, e.g., at the beginning and end of your animation. The range
```

selector is a gray bar in the time axis of your timeline. When you move your mouse over it, six dots will appear allowing you to manipulate it.

</details>
<details>
 <summary>
  File watching does not work on Windows Subsystem for Linux (WSL) 2
 </summary>

When running Vite on WSL2, file system watching does not work if a file is edited by Windows applications.

To fix this, move the project folder into the WSL2 file system and use WSL2 applications to edit files. Accessing the Windows file system from WSL2 is slow, so this will improve performance.

For more information view the
[**Vite Docs**](https://vitejs.dev/config/server-options.html#server-watch).

</details>

[generators]:
 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*
[all]: /api/core/flow#all

File Path: revideo-docs/guide/parameterized-video.mdx
---
sidebar_position: 6
slug: /parameterized-video

Parameterized Videos

To render videos with dynamic inputs or to build video apps, you can use video parameters. Consider this example video:

```tsx editor
import {Txt, makeScene2D} from '@revideo/2d';
import {useScene, createRef} from '@revideo/core';

export default makeScene2D(function* (view) {
  const textRef = createRef<Txt>();

  const name = 'new user';

  view.add(
    <Txt fontSize={1} textWrap={true} ref={textRef} fill={'blue'}>
      Hello {name}!
    </Txt>,
  );

  yield* textRef().scale(30, 2);
});
```

Instead of using the generic `"new user"` string, we might want to use a real name - we can do this using a project variable:

```tsx
import {Txt, makeScene2D} from '@revideo/2d';
import {useScene, createRef} from '@revideo/core';
```

```
export default makeScene2D(function* (view) {
  const textRef = createRef<Txt>();

  const name = useScene().variables.get('username', 'new user');

  view.add(
    <Txt fontSize={1} textWrap={true} ref={textRef} fill={'blue'}>
      Hello {name()}!
    </Txt>,
  );

  yield* textRef().scale(30, 2);
});
```

The first argument of `.get()` refers to the name of the variable we want to use, and the second assigns a default value if no variable is provided. Here, "new user" will be used if the username variable is not provided.

Passing Parameters to `renderVideo()`

You can pass variables to `renderVideo()` as follows:

```ts
import {renderVideo} from '@revideo/renderer';

renderVideo({
  projectFile: './src/project.ts',
  variables: {username: 'Mike'},
});
```

Passing Parameters to visual editor

To use variables in the visual editor, you can pass them to `makeProject` in `src/project.ts`.

```ts
import {makeProject} from '@revideo/core';
import example from './scenes/example?scene';

export default makeProject({
  scenes: [example],
  variables: {username: 'Mike'},
});
```

Complex Parameters

Parameters allow you to customize videos extensively. For instance, you can also pass file names as parameters, which can in turn point to audio or images generated from AI services like text-to-speech software or image generators.

If you want to see an example project using more complex parameters in action, you can check out our
[Youtube Short project](https://github.com/redotvideo/examples/tree/main/youtube-shorts).
This project uses AI-generated images and subtitles, which are passed as a list of words along with timestamps.

File Path: revideo-docs/guide/performance/_category_.yml
label: Performance Considerations
position: 100
link:
  type: generated-index

File Path: revideo-docs/guide/performance/node-parent-reloading.mdx
---
sidebar_position: 1
slug: /node-parent-reloading

Isolating Frequently Changed Nodes

A common use case for Revideo is adding subtitles to a video. Often, this gets handled like this:

```tsx editor
import {Txt, Video, makeScene2D} from '@revideo/2d';
import {useScene, waitFor, createRef} from '@revideo/core';

export default makeScene2D(function* (view) {
 const words = [
  'Here',
  'are',
  'some',
  'subtitles',
  'added',
  'to',
  'the',
  'video',
 ];

 yield view.add(
  <>
   <Video
    src={'https://revideo-example-assets.s3.amazonaws.com/beach-3.mp4'}
    play={true}
    size={['100%', '100%']}
   />
  </>,
 );
```

```
  for (const w of words) {
    const textRef = createRef<Txt>();
    yield view.add(
      <Txt
        fontFamily={'Sans-Serif'}
        fill={'white'}
        fontSize={40}
        ref={textRef}
        text={w}
      />,
    );
    yield* waitFor(0.3);
    textRef().remove();
  }

  yield* waitFor(1);
});
```

Speed up Rendering by not adding `<Txt/>` to `view`

The implementation shown above can be made **significantly faster**, especially when we have more words in our subtitles:

The existing implementation repeatedly modifies the `view` node by always adding
new `<Txt/>` elements to it. All of these operations will cause Revideo to reload the `view` node and therefore also reload the `<Video/>` tag, which takes up a lot of time.

The solution to this is to not add `<Txt/>` elements to the `view` node directly, but to a child container of `view` that is not a parent of our `<Video/>` element. Now, we will not reload all children of `view` during every `<Txt/>` change:

```tsx editor
import {Txt, Video, Layout, makeScene2D} from '@revideo/2d';
```

```
import {useScene, waitFor, createRef} from '@revideo/core';

export default makeScene2D(function* (view) {
 const textContainer = createRef<Layout>();
 const words = [
  'Here',
  'are',
  'some',
  'subtitles',
  'added',
  'to',
  'the',
  'video',
 ];

 yield view.add(
  <>
   <Video
    src={'https://revideo-example-assets.s3.amazonaws.com/beach-3.mp4'}
    play={true}
    size={['100%', '100%']}
   />
   <Layout size={['100%', '100%']} ref={textContainer} />
  </>,
 );

 for (const w of words) {
  const textRef = createRef<Txt>();
  yield textContainer().add(
   <Txt
    fontFamily={'Sans-Serif'}
    fill={'white'}
    fontSize={40}
    ref={textRef}
    text={w}
   />,
  );
```

```
    yield* waitFor(0.3);
    textRef().remove();
  }

  yield* waitFor(1);
});
```

Of course, this does not just apply to `<Txt/>` nodes, but any node that you modify frequently.

File Path: revideo-docs/guide/project-structure.mdx
---
sidebar_position: 2
slug: /project-structure

Project Structure

Revideo projects are structured similar to most Typescript projects. Here is the structure of the default project that gets initialized when you run `npm init @revideo@latest`:

```
my-project/
├─── package.json
├─── package-lock.json
├─── tsconfig.json
├─── vite.config.ts
├─── src/
│    ├─── project.ts
│    ├─── render.ts
│    ├─── project.meta
│    └─── scenes/
│         └─── example.tsx
└─── public/
     └─── my-video.mp4
```

Let's walk through the most relevant files:

`./src/scenes/example.tsx`

Files inside the `./src/scenes` folder such as `example.tsx` are the ones you'll modify the most in order to define video templates. Scenes describe how your video should look like. They need to specify a default export that calls `makeScene2D` on a generator function describing what the desired video should look like:

```tsx
import {Video, makeScene2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const videoFile = useScene().variables.get(
    'video',
    'https://revideo-example-assets.s3.amazonaws.com/stars.mp4',
  )();

  yield view.add(<Video src={videoFile} size={['100%', '100%']} play={true} />);

  yield* waitFor(10);
});
```

`./src/project.ts`

Your project file does two things:

1. It defines an array of scenes to create a full video
2. It accepts video variables that will be passed to your video when you look at it in the visual editor (when you run `npm start`)

```
import {makeProject} from '@revideo/core';

import example from './scenes/example?scene';
import example2 from './scenes/example2?scene';


export default makeProject({
  scenes: [example, example2],
  variables: { video: "https://revideo-example-assets.s3.amazonaws.com/
beach.mp4"}
});
```

```

When specifying multiple scenes, the scenes will be played after another. You can also add [transitions](/transitions) between them. A new scene will not inherit any nodes from an old scene. This can have a positive influence on performance, especially when your scene is very bloated (as it contains a lot of nodes). In this case, recalculating the scene (for instance when navigating to a new time in the player) can be an expensive operation and is faster when you have multiple smaller scenes instead.

However, in most cases, it is not necessary to use multiple scenes. For logical seperation, you can instead define generator function outside of your main generator function in `makeScene2D` and call them there.

### `vite.config.ts`

Revideo projects are served using [vite](https://vitejs.dev/). The `vite.config.ts` file specifically configures the server used to serve the visual editor you see when running `npm start` inside your project. The default config looks as follows:

```ts
import {defineConfig} from 'vite';
import motionCanvas from '@revideo/vite-plugin';

export default defineConfig({
  plugins: [motionCanvas()],
});
```

As you can see, the default settings are sufficient, and the only thing we modify is using the 'motionCanvas' plugin. This plugin enables Motion Canvas / Revideo-related functionality, such as communication between the browser in which the HTML canvas is rendered and drawn to, and a "backend" process running
ffmpeg for audio processing.
```

You can modify some settings of the motion canvas plugin, for instance to point to another project file than the default `project.ts`, or to select another output folder than the default `./output`. You can also modify some vite server settings here, for instance on which port your application is served:

```ts
import {defineConfig} from 'vite';
import motionCanvas from '@revideo/vite-plugin';

export default defineConfig({
  plugins: [
    motionCanvas({
      output: './other-output-folder',
      project: './src/project2.ts',
    }),
  ],
  server: {
    port: 5000,
  },
});
```

`./src/render.ts`

This file contains code to render your video. By default, you can execute its code by running `npm run render` (assuming that you bootstrapped your project using `npm init @revideo@latest`). Note that the [`renderVideo()`](/api/renderer/renderVideo) function accepts variables of its own and does not use the ones from `./src/project.ts`.

```
import {renderVideo} from '@revideo/renderer';

async function render() {
    console.log('Rendering video...');

    // This is the main function that renders the video
```

```
  const file = await renderVideo({
    projectFile: './src/project.ts',
    variables: { video: "" }
    settings: { logProgress: true }
  });

  console.log(`Rendered video to ${file}`);
}

render();
```

This code is not neccessary to start the editor or your project and is not tied to any other rendering functionality (like the renderer of the development server from `npx revideo serve`). You can safely remove it if you don't need it.

Files in `/public`

If you want to work with local files, you can put them into the `/public` folder. You can then access them inside your scene via their name:

```tsx
import {Video, makeScene2D} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  yield view.add(
    <Video src={'/my-video.mp4'} size={['100%', '100%']} play={true} />,
  );

  yield* waitFor(10);
});
```

File Path: revideo-docs/guide/rendering-videos.mdx
---
sidebar_position: 7
slug: /rendering-videos

Rendering Videos

As mentioned in the quickstart guide, you can render videos directly through the user interface of the editor, or through a function call.

Rendering with `renderVideo()` function

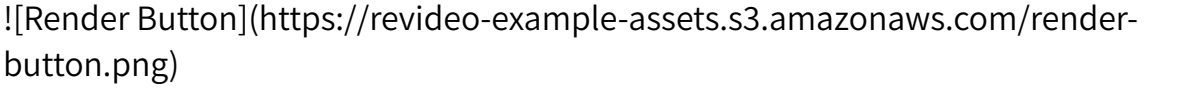
To render videos through a function call, you can use the `renderVideo()` function ([API reference](/api/renderer/renderVideo)):

```ts
import {renderVideo} from '@revideo/renderer';

async function render() {
  console.log('Rendering video...');

  // This is the main function that renders the video
  const file = await renderVideo({
    projectFile: './src/project.ts',
    settings: {logProgress: true},
  });

  console.log(`Rendered video to ${file}`);
}

render();
```

Rendering from the browser

To render videos from the editor, simply click the "Render" Button after opening

the editor with `npm start`:



How does Rendering work?

The rendering process mostly runs in the browser. The only part that runs in a separate "backend" nodejs process is everything related to audio processing.

The code in the browser loops through all frames in the defined video, draws the defined frames onto an
[HTML Canvas](https://www.w3schools.com/html/html5_canvas.asp) and feeds them
into the `VideoEncoder` of the
[WebCodecs API](https://developer.mozilla.org/en-US/docs/Web/API/VideoEncoder).
This turns all frames into a mute mp4 file.

In the "backend", ffmpeg is used to extract all audio from the `<Video/>` and `<Audio/>` elements of the project and to merge them into a single audio file. This file finally gets merged with the mute video produced in the browser to obtain the resulting mp4 video.

Rendering Speeds

Since the release of v0.4.6, rendering videos is almost always faster than real-time (meaning that rendering a 1-minute video takes less than one minute). We have created a [guide](/common-issues/slow-rendering) that describes what affects rendering speeds and how to make rendering faster.

Parallelized Rendering

If you want to speed up rendering, you can parallelize the rendering process of your videos. This means that instead of rendering your full video through a single process, you can use multiple processes to render only small parts of your video and subsequently merge the single parts together (for example, you

can use 10 processes to render 1 minute of a 10-minute video each).

The `renderVideo()` function
([API reference](http://localhost:3000/renderer/renderVideo)) provides a
`settings.worker` argument that you can use to parallelize rendering in a single process. This is useful when you have a lot of RAM available.

Alternatively, the best way to set up parallelized rendering is to use
serverless function providers like AWS Lambda. Revideo provides a
`renderPartialVideo()` function
([API reference](http://localhost:3000/renderer/renderPartialVideo)) that you can use to render a partial video - you can use it together with
`concatenateMedia()` ([API reference](/ffmpeg/concatenateMedia)) and
`mergeAudioWithVideo()`
([API reference](http://localhost:3000/ffmpeg/mergeAudioWithVideo)) to parallelize rendering across serverless functions. We also provide guides and example projects for setting up parallelized rendering on
[AWS Lambda](https://github.com/redotvideo/examples/tree/main/parallelized-aws-lambda)
(recommended) and
[Google Cloud Functions](https://github.com/redotvideo/examples/tree/main/google-cloud-run-parallelized).

File Path: revideo-docs/intro.md
---
sidebar_position: 1
slug: /

Welcome to Revideo!

Revideo is an open-source framework for programmatic video editing. It lets you create video templates in Typescript and provides an API to render these video templates with dynamic inputs. It also provides a player component that you can embed into your website to let users preview videos before exporting them to mp4. Developers use Revideo to automate certain video editing tasks or to build entire web-based video editors.

Revideo is forked from
[Motion Canvas](https://github.com/motion-canvas/motion-canvas), an editor that
lets you create animations with code. While Motion Canvas happens to be distributed as an npm package, the maintainers don't intend for it to be used as a library but as a standalone editing tool.

File Path: revideo-docs/motion-canvas/_category_.yml
label: Motion Canvas Guide
position: 3
link:
  type: generated-index

File Path: revideo-docs/motion-canvas/code/tsconfig.json

```
{
  "extends": "@revideo/2d/tsconfig.project.json",
  "include": ["."]
}
```

File Path: revideo-docs/motion-canvas/code-animations.mdx
---
sidebar_position: 12
slug: /code-animations

Code Animations

```tsx editor mode=preview
import {makeScene2D, Code} from '@revideo/2d';
import {all, createRef, DEFAULT, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
 const code = createRef<Code>();

 view.add(
  <Code
   ref={code}
   fontSize={28}
   fontFamily={'JetBrains Mono, monospace'}
   offsetX={-1}
   x={-400}
   code={'const number = 7;'}
  />,
 );

 yield* waitFor(0.6);
 yield* all(
  code().code.replace(code().findFirstRange('number'), 'variable', 0.6),
  code().code.prepend(0.6)`function example() {\n  `,
  code().code.append(0.6)`\n}`,
 );

 yield* waitFor(0.6);
 yield* code().selection(code().findFirstRange('variable'), 0.6);

 yield* waitFor(0.6);
```

```
  yield* all(
    code().code('const number = 7;', 0.6),
    code().selection(DEFAULT, 0.6),
  );
});
```

The [`Code`](/api/2d/components/Code) node is used for displaying code snippets.
It supports syntax highlighting and a handful of different methods for animating the code.

Parsing and Highlighting

First things first, if you just copy any of the snippets in this tutorial you'll notice that the displayed code has a uniform color. The default highlighter uses [`Lezer`](https://lezer.codemirror.net/) to parse and highlight the code but to do that it needs the grammar for the language you're using. You can set that up in your project configuration file.

For this tutorial, you should install the `javascript` grammar:

```bash
npm i @lezer/javascript
```

Then, in your project configuration, instantiate a new `LezerHighlighter` using the imported grammar, and set it as the default highlighter:

```ts title="src/project.ts"
import {makeProject} from '@revideo/core';
import example from './scenes/example?scene';

// highlight-start
import {Code, LezerHighlighter} from '@revideo/2d';
import {parser} from '@lezer/javascript';
```

````
Code.defaultHighlighter = new LezerHighlighter(parser);
// highlight-end

export default makeProject({
  scenes: [example],
});
```

Now all `Code` nodes in your project will use `@lezer/javascript` to parse and highlight the snippets. If you want to use more than one language, check out the [Multiple Languages](#multiple-languages) section.

:::info

Note that, by default, the JavaScript parser doesn't support JSX or TypeScript. You can enable support for these via [dialects][dialects]. The dialects available for a given parser are usually listed in the documentation of the grammar package.

```ts
Code.defaultHighlighter = new LezerHighlighter(
  parser.configure({
    // Provide a space-separated list of dialects to enable:
    dialect: 'jsx ts',
  }),
);
```

:::

## Defining Code

The code to display is set via the [`code`](/api/2d/components/Code#code) property. In the simplest case, you can just use a string:

```tsx editor
import {makeScene2D, Code} from '@revideo/2d';
````

````
export default makeScene2D(function* (view) {
  view.add(
    // prettier-ignore
    <Code
      fontSize={28}
      code={'const number = 7;'}
    />,
  );
});
```

However, usually code snippets contain multiple lines of code. It's much more convenient to use a template string for this (denoted using the backtick character `` ` ``):

```tsx
view.add(
  <Code
    fontSize={28}
    // highlight-start
    code={`\
function example() {
  const number = 7;
}
`}
    // highlight-end
  />,
);
```

Notice two things here:

- The code snippet ignores the indentation of the template string itself. The template string preserves all whitespace characters, so any additional spaces or tabs at the beginning of each line would be included in the snippet.

- The backslash character (`\`) at the very beginning is used to escape the first newline character. This lets the snippet start on a new line without actually including an empty line at the beginning. Without the slash, the equivalent code would have to be written as:

  ```tsx
  view.add(
    <Code
      fontSize={28}
      // highlight-start
      code={`function example() {
    const number = 7;
  }
  `}
      // highlight-end
    />,
  );
  ```

Template strings allow you to easily include variables in your code snippets with the `${}` syntax. In the example below, `${name}` is replaced with the value of the `name` variable (which is `number` in this case):

```tsx
// highlight-next-line
const name = 'number';

view.add(
  <Code
    fontSize={28}
    code={`\
function example() {
  // highlight-next-line
  const ${name} = 7;
}
`}
  />,
);
```

```

Any valid JavaScript expression inside the `${}` syntax will be included in the code snippet:

```tsx
// highlight-next-line
const isRed = true;

view.add(
  <Code
    fontSize={28}
    code={`\
function example() {
  // highlight-next-line
  const color = '${isRed ? 'red' : 'blue'}';
}
`}
  />,
);
```

## Using Signals

If you try to use signals inside the `${}` syntax, you'll notice that they don't work as expected. Invoking a signal inside a template string uses its current value and then never updates the snippet again, even if the signal changes:

```tsx editor mode=code
import {makeScene2D, Code} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const nameSignal = Code.createSignal('number');
  view.add(
    // prettier-ignore
    <Code

```
    fontSize={28}
    code={`const ${nameSignal()} = 7;`}
   />,
  );

  yield* waitFor(1);
  nameSignal('newValue');
  // The code snippet still displays "number" instead of "newValue".
  yield* waitFor(1);
});
```

Trying to pass the signal without invoking it is even worse. Since each signal is a function, it will be stringified and included in the snippet:

```tsx editor
import {makeScene2D, Code} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const nameSignal = Code.createSignal('number');
  view.add(
    // prettier-ignore
    <Code
      fontSize={28}
      code={`const ${nameSignal} = 7;`}
    />,
  );

  yield* waitFor(1);
  nameSignal('newValue');
  yield* waitFor(1);
});
```

This happens because template strings are parsed immediately when our code is executed. To work around this, you can use a custom [tag function][tag-function]

called [`CODE`](/api/2d/code#CODE). It allows the `Code` node to parse the template string in a custom way and correctly support signals. It's really easy to use, simply put the `CODE` tag function before your template string:

```tsx editor
import {makeScene2D, Code, CODE} from '@revideo/2d';
import {waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const nameSignal = Code.createSignal('number');
  view.add(
    <Code
      fontSize={28}
      // Note the CODE tag function here:
      code={CODE`const ${nameSignal} = 7;`}
    />,
  );

  yield* waitFor(1);
  nameSignal('newValue');
  // Now the code snippet is updated accordingly.
  yield* waitFor(1);
});
```

The value returned by `CODE` can itself be nested in other template strings:

```tsx
const implementation = CODE`\
    console.log('Hello!');
    return 7;`;

const method = CODE`\
  greet() {
${implementation}
  }`;
```

````
const klass = CODE`\
class Example {
${method}
}
`;

view.add(<Code code={klass} />);
// class Example {
//   greet() {
//   console.log('Hello!');
//   return 7;
//   }
// }
```

You might have noticed that these examples used a specialized type of signal created using [`Code.createSignal()`](/api/2d/components/Code#createSignal). While the generic [`createSignal()`](/api/core/signals#createSignal) would work fine in these simple examples, the specialized signal will shine once you start animating your code snippets.

Animating Code

The `Code` node comes with a few different techniques for animating the code depending on the level of control you need.

Diffing

The default method for animating code is diffing. It's used whenever you tween the `code` property:

```tsx editor
import {makeScene2D, Code} from '@revideo/2d';
import {createRef} from '@revideo/core';

export default makeScene2D(function* (view) {
  const code = createRef<Code>();
````

```
  view.add(
    <Code
      ref={code}
      fontSize={28}
      offsetX={-1}
      x={-400}
      code={`\
function example() {
  const number = 9;
}`}
    />,
  );

  yield* code().code('const nine = 9;', 0.6).wait(0.6).back(0.6).wait(0.6);
});
```

This method uses the patience diff algorithm to determine the differences
between the old and new code snippets. It then animates the changes accordingly.

`append` and `prepend`

For cases where you want to add some code at the beginning or end of the
snippet, you can use the [`append`](/api/2d/code/CodeSignalContext#append)
and
[`prepend`](/api/2d/code/CodeSignalContext#prepend) methods. They can
either
modify the code immediately or animate the changes over time:

```tsx editor
import {makeScene2D, Code} from '@revideo/2d';
import {createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const code = createRef<Code>();
```

```
  view.add(
    // prettier-ignore
    <Code
      ref={code}
      fontSize={28}
      offsetX={-1}
      x={-400}
    />,
  );

  // append immediately
  code().code.append(`const one = 1;`);

  // animate using the signal signature
  yield* code().code.append('\nconst two = 2;', 0.6);

  // animate using the template tag signature
  yield* code().code.append(0.6)`
const three = 3;`;

  // prepend works analogically
  yield* code().code.prepend('// example\n', 0.6);

  yield* waitFor(0.6);
});
```

`insert`, `replace`, and `remove`

For more granular control over the changes, you can use
[`insert`](/api/2d/code/CodeSignalContext#insert),
[`replace`](/api/2d/code/CodeSignalContext#replace), and
[`remove`](/api/2d/code/CodeSignalContext#remove) to modify the code at specific
points. Check out [Code Ranges](#code-ranges) for more information on how to specify points in your code snippets.

```tsx editor
import {makeScene2D, Code, word, lines} from '@revideo/2d';
import {all, createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const code = createRef<Code>();

  view.add(
    <Code
      ref={code}
      fontSize={28}
      offsetX={-1}
      x={-400}
      code={`\
function example() {
  console.log('Hello!');
}`}
    />,
  );

  // insert code at line 2, column 0
  yield* code().code.insert([2, 0], '  return 7;\n', 0.6);

  // replace the word "Hello!" with "Goodbye!"
  yield* code().code.replace(word(1, 15, 6), 'Goodbye!', 0.6);

  // remove line 2
  yield* code().code.remove(lines(2), 0.6);

  // animate multiple changes at the same time
  yield* all(
    code().code.replace(word(0, 9, 7), 'greet', 0.6),
    code().code.replace(word(1, 15, 8), 'Hello!', 0.6),
  );

  yield* waitFor(0.6);
});
```

```

### `edit`

The [`edit`](/api/2d/code/CodeSignalContext#edit) method offers a different way of defining code transitions. It's used together with the [`replace`](/api/2d/code#insert), [`insert`](/api/2d/code#insert), and [`remove`](/api/2d/code#remove) helper functions that are inserted into the template string. They let you specify the changes in a more visual way, without having to know the exact positions in the code:

```tsx editor
import {makeScene2D, Code, replace, insert, remove} from '@revideo/2d';
import {createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const code = createRef<Code>();

  view.add(
    // prettier-ignore
    <Code
      ref={code}
      fontSize={28}
      offsetX={-1}
      x={-400}
    />,
  );

  yield* code().code.edit(0.6)`\
function example() {
  ${insert(`// This is a comment
  `)}console.log("${replace('Hello!', 'Goodbye!')}");
${remove(`  return 7;
`)}}`;

  yield* waitFor(0.6);
});
```
```

```

### Signals

Notice that all the methods used above are not invoked on the `Code` node but rather on its `code` property. It may seem unnecessarily verbose but there's a good reason for it: the `code` property is a specialized code signal, just like the ones created by [`Code.createSignal()`](/api/2d/components/Code#createSignal). This means that all the animation methods are also available on your own signals:

```tsx editor
import {makeScene2D, Code, CODE} from '@revideo/2d';
import {all, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const message = Code.createSignal(`Hello, world!`);
  const body = Code.createSignal(CODE`console.log('${message}');`);

  view.add(
    <Code
      fontSize={28}
      offsetX={-1}
      x={-400}
      code={CODE`\
function hello() {
  ${body}
}`}
    />,
  );

  yield* waitFor(0.3);
  // prettier-ignore
  yield* all(
    message('Goodbye, world!', 0.6),
    body.append(0.6)`\n  return 7;`,
  );
```
```

```
  yield* waitFor(0.3);
});
```

Code signals can also be nested in the template strings passed to the animation methods:

```tsx editor mode=code
import {makeScene2D, Code, CODE} from '@revideo/2d';
import {createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const code = createRef<Code>();

  view.add(
    // prettier-ignore
    <Code
      ref={code}
      fontSize={28}
      offsetX={-1}
      x={-400}
      code={'// example'}
    />,
  );

  const body = Code.createSignal(CODE`console.log('Hello!');`);
  yield* waitFor(0.3);
  // Notice how the CODE tag is not used here because
  // "append" already supports nested signals:
  yield* code().code.append(0.6)`
function hello() {
  ${body}
}`;

  // The "body" signal remains reactive after the append animation:
  yield* body(`console.log('Goodbye!');`, 0.6);
  yield* waitFor(0.3);
```

});
```

## Code Ranges

A [`CodeRange`](/api/2d/code#CodeRange) is used to specify a continuous span of
characters using line and column numbers. It can be used for editing the code, visually selecting a part of it, or querying the positions and sizes of characters.

Code ranges have the following structure:

```js
// prettier-ignore
[[startLine, startColumn], [endLine, endColumn]];
```

For example, to select the first three characters of the second line, you would use the following range:

```js
// prettier-ignore
[[1, 0], [1, 3]];
```

Keep in mind that both lines and columns are zero-based. Additionally, you should think of columns as being located on the left side of the characters, meaning that if you want to include the character at column `n` you should use `n + 1` as the end column.

For convenience, the [`word`](/api/2d/code#word) and
[`lines`](/api/2d/code#lines) helper functions are provided to create some of the common types of ranges:

```ts
// a range starting at line 1, column 3,
```
```

```
// spanning 3 characters:
word(1, 3, 3);

// a range starting at line 1, column 3,
// spanning until the end of the line:
word(1, 3);

// a range containing lines from 1 to 3 (inclusive):
lines(1, 3);

// a range containing line 2
lines(2);
```

Once you create a `Code` node, you can use its
[`findFirstRange`](/api/2d/components/Code#findFirstRange),
[`findAllRanges`](/api/2d/components/Code#findAllRanges), and
[`findLastRange`](/api/2d/components/Code#findLastRange) methods to find the ranges that contain a specific string or match the given regular expression:

```tsx editor mode=code
import {makeScene2D, Code} from '@revideo/2d';
import {createRef, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
  const code = createRef<Code>();

  view.add(
    <Code
      ref={code}
      fontSize={28}
      offsetX={-1}
      x={-400}
      code={`\
function example() {
  console.log('Hello!');
}`}
```

```
  />,
 );

 yield* code().code.replace(
  // find the range of "example" and replace it with "greet"
  code().findFirstRange('example'),
  'greet',
  0.6,
 );

 yield* waitFor(0.6);
});
```

Code Selection

The [`selection`](/api/2d/code/CodeSignalContext#selection) property can be used
to visually distinguish a part of the code snippet. The selection is specified
using an individual [`code range`](#code-ranges) or an array of ranges:

```tsx editor
import {makeScene2D, Code, lines} from '@revideo/2d';
import {createRef, DEFAULT, waitFor} from '@revideo/core';

export default makeScene2D(function* (view) {
 const code = createRef<Code>();

 view.add(
  <Code
   ref={code}
   fontSize={28}
   offsetX={-1}
   x={-400}
   code={`\
function hello() {
 console.log('Hello');
```

```
}`}
    />,
  );

  // select all instances of "hello" (case-insensitive)
  yield* code().selection(code().findAllRanges(/hello/gi), 0.6);
  yield* waitFor(0.3);

  // select line 1
  yield* code().selection(lines(1), 0.6);
  yield* waitFor(0.3);

  // reset the selection
  yield* code().selection(DEFAULT, 0.6);
  yield* waitFor(0.3);
});
```

Querying Positions and Sizes

[`getPointBBox`](/api/2d/components/Code#getPointBBox) and
[`getSelectionBBox`](/api/2d/components/Code#getSelectionBBox) can be used
to
retrieve the position and size of a specific character or a range of characters,
respectively. The returned value is a [`bounding box`](/api/core/types/BBox) in
the local space of the `Code` node.

The following example uses `getSelectionBBox` to draw a rectangle around the
word `log`:

```tsx editor
import {Code, Rect, makeScene2D} from '@revideo/2d';
import {createRef, createSignal} from '@revideo/core';

export default makeScene2D(function* (view) {
  const code = createRef<Code>();
```

```
  view.add(
    <Code
      ref={code}
      fontSize={28}
      offsetX={-1}
      x={-400}
      code={`\
function hello() {
  console.log('Hello');
}`}
    />,
  );

  const range = createSignal(() => {
    const range = code().findFirstRange('log');
    const bboxes = code().getSelectionBBox(range);
    // "getSelectionBBox" returns an array of bboxes,
    // one for each line in the range. You can just
    // use the first one for this example.
    const first = bboxes[0];
    return first.expand([4, 8]);
  });

  code().add(
    <Rect
      offset={-1}
      position={range().position}
      size={range().size}
      lineWidth={4}
      stroke={'white'}
      radius={8}
    />,
  );
});
```

Custom Themes

`LezerHighlighter` uses CodeMirror's
[`HighlightStyle`](https://codemirror.net/examples/styling/) to assign colors to specific code tokens. By default, the
[`DefaultHighlightStyle`](/api/2d/code#DefaultHighlightStyle) is used. You can specify your own style by passing it as the second argument to the
`LezerHighlighter` constructor:

```ts
import {Code, LezerHighlighter} from '@revideo/2d';
import {HighlightStyle} from '@codemirror/language';
import {tags} from '@lezer/highlight';
import {parser} from '@lezer/javascript';

const MyStyle = HighlightStyle.define([
  {tag: tags.keyword, color: 'red'},
  {tag: tags.function(tags.variableName), color: 'yellow'},
  {tag: tags.number, color: 'blue'},
  {tag: tags.string, color: 'green'},
  // ...
]);

Code.defaultHighlighter = new LezerHighlighter(parser, MyStyle);
```

Multiple Languages

You can configure highlighters on a per-node basis using the
[`highlighter`](/api/2d/components/Code#highlighter) property. This will override the default highlighter set in the project configuration file:

```tsx
import {Code, LezerHighlighter} from '@revideo/2d';
import {parser} from '@lezer/rust';

const RustHighlighter = new LezerHighlighter(parser);
```

````
// ...

view.add(
  <Code
    // this node uses the default parser
    offsetX={-1}
    x={-400}
    code={`
function hello() {
  console.log('Hello!');
}
`}
  />,
);

view.add(
  <Code
    // this node uses the Rust parser
    highlighter={RustHighlighter}
    offsetX={1}
    x={400}
    code={`
fn hello() {
  println!("Hello!");
}
`}
  />,
);
```

It can be useful to create a custom component for the languages you often use. You can use the [`withDefaults`](/api/2d/utils#withDefaults) helper function to quickly extend any node with your own defaults:

```ts title="src/nodes/RustCode.ts"
import {Code, LezerHighlighter, withDefaults} from '@revideo/2d';
import {parser} from '@lezer/rust';
````

````
const RustHighlighter = new LezerHighlighter(parser);

export const RustCode = withDefaults(Code, {
  highlighter: RustHighlighter,
});
```

```tsx title="src/scenes/example.tsx"
import {RustCode} from '../nodes/RustCode';

// ...

view.add(
  <RustCode
    code={`
fn hello() {
  println!("Hello!");
}
`}
  />,
);
```
````
````

File Path: revideo-docs/motion-canvas/configuration.mdx
---
sidebar_position: 16
slug: /configuration

Configuration

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Motion Canvas is configured in the Vite's configuration file:

```ts title="vite.config.ts"
import {defineConfig} from 'vite';
import motionCanvas from '@revideo/vite-plugin';

export default defineConfig({
  plugins: [
    motionCanvas({
      // custom options
    }),
  ],
});
```

`project`

- Type: `string | string[]`
- Default: `./src/project.ts`

The import path of the project file, relative to the configuration file. Globs are also supported. A project file must contain a default export with an instance of the `Project` class.

When set to an array, the Editor will display a project selection screen, making it possible to change the project without restarting Vite.

```ts
motionCanvas({
  project: [
    // highlight-start
    './src/firstProject.ts',
    './src/secondProject.ts',
  ],
});
```

`output`

- Type: `string`
- Default: `./output`

An output path to which the animation will be saved. Relative to the configuration file.

When rendering, the complete path has the following format:

```
[output]/[projectName]/[frameNumber].[extension]
```

Stills are saved to a `still` subdirectory:

```
[output]/still/[projectName]/[frameNumber].[extension]
```

`bufferedAssets`

- Type: RegExp | false
- Default: `/\.(wav|ogg)$/`

Defines which assets should be buffered before being sent to the browser.

Streaming larger assets directly from the drive may cause issues with other applications. For instance, if an audio file is being used in the project, Adobe Audition will perceive it as "being used by another application" and refuse to override it.

Buffered assets are first loaded to the memory and then streamed from there. This leaves the original files open for modification with hot module replacement still working.

`editor`

- Type: `string`
- Default: `@revideo/ui`

The import path of the editor package.

This option can be used to develop/use an editor different than the official one. The path will be resolved using Node.js module resolution rules. It should lead to a directory containing the following files:

- `editor.html` - The HTML template for the editor.
- `styles.css` - The editor styles.
- `main.js` - A module exporting necessary factory functions.

`main.js` should export the following functions:

- `editor` - Receives the project factory as its first argument and creates the user interface.
- `index` - Receives a list of all projects as its first argument and creates the initial page for selecting a project.

`proxy`

- Type: `boolean | MotionCanvasCorsProxyOptions`
- Default: `false`

The configuration of the Proxy used for remote sources.

This passes configuration to Motion Canvas' proxy. The proxy is disabled by default. You can either pass `true` or a config object to enable it.

- `allowedMimeTypes?: string[]` - Set which types of resources are allowed by default. Catchall on the right side is supported. Pass an empty Array to allow all types of resources, although this is not recommended. Defaults to `["image/*", "video/*"]`.
- `allowListHosts?: string[]` - Set which hosts are allowed Note that the host is everything to the left of the first `/`, and to the right of the protocol `https://`. AllowList is not used by default, although you should consider setting up just the relevant hosts.

File Path: revideo-docs/motion-canvas/custom-components.mdx
---
sidebar_position: 9
slug: /custom-components
---

import AnimationPlayer from '@site/src/components/AnimationPlayer';
import CodeBlock from '@theme/CodeBlock';
import source from '!!raw-loader!@revideo/examples/src/components/Switch';

Custom Components

Components are classes like [`Rect`](/api/2d/components/Rect) and [`Circle`](/api/2d/components/Circle) that can abstract rendering and data functionality into reusable, modular pieces. To use a component in a scene, add it to the view and provide arguments to the component.

```tsx
<Switch initialState={false} />
```

To define what arguments a component will take, first define an interface. All properties of the interface must be wrapped in [`SignalValue<>`](/api/core/signals#SignalValue) as such:

```tsx
//  You can extend an existing props interface
//  such as LayoutProps, ShapeProps or NodeProps to
//  include their properties alongside the ones you
//  define

export interface SwitchProps extends NodeProps {
  initialState?: SignalValue<boolean>;

  // We don't use color here because we want
  // to be able to pass hex strings and rgb
  // values to accent rather than a `Color`
```

```
  accent?: SignalValue<PossibleColor>;
}
```

Next, create a class for your components. The component class must extend [`Node`](/api/2d/components/Node) or one of its subclasses. If you don't want to inherit any methods from an existing component, extend your class from `Node`. We advise extending from the component most similar to the component you are building. For instance, if you were to make a component including a [`Layout`](/api/2d/components/Layout), you should extend [`Layout`](/api/2d/components/Layout) and [`LayoutProps`](/api/2d/components/LayoutProps).

```tsx
export interface SwitchProps extends NodeProps {
  // properties
}

export class Switch extends Node {
  // implementation
}
```

To use the properties defined in the interface, your class _must_ contain a property with the same name. Motion Canvas provides type decorators to facilitate this like `@initial()` and `@signal()`. Click [here](/signals) for more information on signals.

Here is an example of how you would define such properties:

```tsx
export class Switch extends Node {
  // @initial - optional, sets the property to an
  // initial value if it was not provided.
  @initial(false)
  // @signal - is required by motion canvas
  // for every prop that was passed in.
```

```
  @signal()
  public declare readonly initialState: SimpleSignal<boolean, this>;

  @initial('#68ABDF')
  // @colorSignal - some complex types provide a dedicated decorator for
  // signals that takes care of parsing.
  // In this case, `accent` will automatically convert strings into `Color`s
  @colorSignal()
  public declare readonly accent: ColorSignal<this>;
  // ...
}
```

Notice how colors are wrapped in `ColorSignal<>` while any other type (even user-defined ones) are wrapped in `SimpleSignal<>`. The type does not need to be
passed to color signal as Motion Canvas knows that it must be of a color-resolvable type. In both, the class is passed at the end of the wrapper to register the signal to the class. Properties must be initialised with the `public`, `declare` and `readonly` keywords.

Normal properties can be defined as normal. For example:

```tsx
export class Switch extends Node {
  public constructor(props?: SwitchProps) {
    super({
      // If you wanted to ensure that layout was always
      // true for this component, you could add it here
      // as such:
      // layout: true
      ...props,
    });
    // ...
  }
}
```

The `props` parameter can also be useful outside the `super()` call to access your data elsewhere. For example, if you were building a component to display an array, you could use props to set the color of every [Rect](/api/2d/components/Rect) in the array.

Now we can add elements to the view by using `this.add()`, much like you would add to a scene's view:

```tsx
export class Switch extends Node {
  public constructor(props?: SwitchProps) {
    // ...
    this.add(
      <Rect>
        <Circle />
      </Rect>,
    );
  }
}
```

Since this is a class, you can also add methods. This is especially useful when wanting to animate a component easily. Here is an example of a method for toggling our switch:

```tsx
export class Switch extends Node {
  // ...
  public *toggle(duration: number) {
    yield* all(
      tween(duration, value => {
        // ...
      }),
      tween(duration, value => {
        // ...
      }),
```

```
    );
    this.isOn = !this.isOn;
  }
}
```

Here is the source code for the component we have built throughout this guide:

<CodeBlock language="tsx">{source}</CodeBlock>

File Path: revideo-docs/motion-canvas/custom-font.mdx
---
sidebar_position: 20
slug: /custom-font

Custom font

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Loading fonts from the web

To use a font from hosters like Google Fonts. First make a css file under `src`

```diff
root
 └──src
   ├──scenes/
   ├──revideo.d.ts
   ├──project.meta
   ├──project.ts
+  └──global.css
```

Inside `global.css`, import the font using `@import url(your link)`.

```css
@import url('https://fonts.googleapis.com/css2?
family=Fira+Code:wght@400;700&display=swap');
```

Then, in `project.ts`, import the css file.

```ts
import {makeProject} from '@revideo/core';
```

```
import example from './scenes/example?scene';

import './global.css'; // <- import the css

export default makeProject({
  scenes: [example],
});
```

Now you can reference the fonts in the `fontFamily` property in this project.

```tsx
<Txt fontFamily={'Fira Code'}>Fira Code</Txt>
```

Loading fonts from local

For local fonts, make a directory `fonts` and put your font inside it.

```diff
root
 └──public
+ └──fonts *
+   └──CASCADIACODE.TTF
```

Inside `global.css`, import the font using `@font-face`.

```css
@import url('https://fonts.googleapis.com/css2?
family=Fira+Code:wght@400;700&display=swap');

@font-face {
  font-family: 'Cascadia Code';
  src:
    local('Cascadia Code'),
    url(public/fonts/CASCADIACODE.TTF) format('truetype');
```

}
```

Notice the name of the font will match the string in `@font-face/font-family` from the css.

```tsx
<Layout direction={'column'} alignItems={'center'} layout>
  <Txt fontFamily={'Fira Code'}>Fira Code</Txt>
  <Txt fontFamily={'Cascadia Code'}>Cascadia Code</Txt>
</Layout>
```
```

File Path: revideo-docs/motion-canvas/experimental.mdx
---
sidebar_position: 10
slug: /experimental

Experimental features

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Motion Canvas follows the [semver](https://semver.org/) versioning scheme. Among
other things, this means that any release with the same major version will be backwards compatible. For example, an animation made using `v3.2.0` (In this case the major version is `3`) will work just fine with `v3.2.1`, `v3.3.0`, `v3.4.0`, and so on.

Experimental features don't follow this rule. They are subject to change at any time and may break your animations. They are meant to be used for testing and feedback purposes only.

:::experimental

Throughout the documentation, experimental features are marked with a warning like this one.

:::

In order to use experimental features, you need to enable them in your project configuration:

```ts title="project.ts"
export default makeProject({
  // highlight-next-line
  experimentalFeatures: true,
  // ...
```

```
});
```

File Path: revideo-docs/motion-canvas/filters-and-effects.mdx
---
sidebar_position: 19
slug: /filters-and-effects
---

import Fiddle from '@site/src/components/Fiddle';
import filtersPreview from '!!raw-loader!./code/filters-and-effects/filters-preview.tsx';
import filtersOrder from '!!raw-loader!./code/filters-and-effects/filters-order.tsx';
import filtersMaskingVisualizedSourceIn from '!!raw-loader!./code/filters-and-effects/masking-visualized-source-in.tsx';

Filters and Effects

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Because Motion Canvas is built on top of the Browser's 2D Rendering Context, we can make use of several canvas operations that are provided by the Browser.

Filters

Filters let you apply various effects to your nodes. You can find all available filters on
[MDN](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/filter).

<Fiddle mode="preview">{filtersPreview}</Fiddle>

Every node has a `filters` property containing an array of [filters](/api/2d/partials#Functions) that will be applied to the node. You can declare this array yourself, or use the `filters` property to configure individual filters. Both ways are shown in the following example:

:::info

Some filters, like `opacity` and `drop-shadow`, have their own dedicated properties directly on the [`Node`](/api/2d/components/Node#opacity), class.

:::

```tsx editor
// snippet Filters Property
import {Img, makeScene2D} from '@revideo/2d';
import {createRef} from '@revideo/core';

export default makeScene2D(function* (view) {
  view.fill('#141414');

  const iconRef = createRef<Img>();
  yield view.add(<Img src={'/img/logo_dark.svg'} size={200} ref={iconRef} />);
  // Modification happens by accessing the `filters` property.
  // Individual filters don't need to be initialized. If a filter you set doesn't
  // exists, it will be automatically created and added to the list of filters.
  // If you have multiple filters of the same type, this will only
  // modify the first instance (you can use the array method for more control).
  yield* iconRef().filters.blur(10, 1);
  yield* iconRef().filters.blur(0, 1);
});

// snippet Filters Array
import {makeScene2D, Img, blur} from '@revideo/2d';
import {createSignal} from '@revideo/core';

export default makeScene2D(function* (view) {
  view.fill('#141414');

  const blurSignal = createSignal(0);
  yield view.add(
    <Img
      src={'/img/logo_dark.svg'}
      size={200}
      /* Modification happens by changing the Filters inside the 'filters' array */
```

```
    filters={[blur(blurSignal)]}
   />,
  );
  yield* blurSignal(10, 1);
  yield* blurSignal(0, 1);
});
```

Keep in mind that the order in which you apply the effects does matter, as can be seen in the following example:

```
<Fiddle mode="preview" ratio={['3']}>
 {filtersOrder}
</Fiddle>
```

Masking and composite operations

Composite operations define how the thing we draw (source) interacts with what is already on the canvas (destination). Among other things, it allows us to define complex masks. MDN has a
[great visualisation of all available composite operations](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/globalCompositeOperation#operations).

You can create a mask by treating one node as the "masking" / "stencil" layer, and another node as the "value" layer. The mask layer will define if the value layer will be visible or not. The value layer will be what's actually visible in the end.

```
<Fiddle mode="preview" ratio={['3']}>
 {filtersMaskingVisualizedSourceIn}
</Fiddle>
```

Any of the following composite operations can be used to create a mask: `source-in`, `source-out`, `destination-in`, and `destination-out`. There is also a `xor` operation which can be helpful if you want two value layers that hide each other on overlap. Use the dropdown below to browse all examples.

```
import sourceInExample from '!!raw-loader!./code/filters-and-effects/source-in-example.tsx';
import sourceOutExample from '!!raw-loader!./code/filters-and-effects/source-out-example.tsx';
import destinationInExample from '!!raw-loader!./code/filters-and-effects/destination-in-example.tsx';
import destinationOutExample from '!!raw-loader!./code/filters-and-effects/destination-out-example.tsx';
import xorExampleExample from '!!raw-loader!./code/filters-and-effects/xor-example.tsx';
import xorExampleWithSourceInExample from '!!raw-loader!./code/filters-and-effects/xor-destination-in-example.tsx';

<Fiddle>
 {'// snippet source-in\n' +
   sourceInExample +
   '\n// snippet source-out\n' +
   sourceOutExample +
   '\n// snippet destination-in\n' +
   destinationInExample +
   '\n// snippet destination-out\n' +
   destinationOutExample +
   '\n// snippet xor\n' +
   xorExampleExample +
   '\n//snippet xor and source-in\n' +
   xorExampleWithSourceInExample}
</Fiddle>

## Cached nodes

Both filters and composite operations require a cached [`Node`](/api/2d/components/Node). Filters can set it automatically, while composite operations require you to set it explicitly on an ancestor [`Node`](/api/2d/components/Node) (usually the parent node).

A cached [`Node`](/api/2d/components/Node) and its children are rendered on an
```

offscreen canvas first, before getting added to the main scene.
For filters this is needed because they are applied to the entire canvas. By creating a new canvas and moving the elements that should get affected by the filters over, applying filters to the entire "new" canvas, and then moving back the result, you effectively only apply the filters to the moved elements.

To turn a [`Node`](/api/2d/components/Node) into a cached node, simply pass the
[`cache`](/api/2d/components/NodeProps/#cache) property

```tsx
<Node cache>...</Node>
// or
<Node cache={true}>...</Node>
```

All components inherit from [`Node`](/api/2d/components/Node), so you can set the cache on all of them.

File Path: revideo-docs/motion-canvas/flow.mdx
---
sidebar_position: 2
slug: /flow
---

import ApiSnippet from '@site/src/components/Api/ApiSnippet';

Animation flow

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Motion Canvas uses generator functions to describe animations.

A generator function is a function that can return multiple values:

```ts
function* example() {
  yield 1;
  yield 2;
  yield 3;
}

const generator = example();

console.log(generator.next().value); // 1;
console.log(generator.next().value); // 2;
console.log(generator.next().value); // 3;
```

When the `yield` keyword is encountered, the execution of the function pauses, and resumes only when the caller requests another value. This is particularly useful when declaring animations - usually we want to change the things on the screen in incremental steps to create an illusion of movement. We also want to wait a constant amount of time between these updates so that our eyes can register what's happening. With generators, we can update things in-between the

`yield` keywords, and then wait for a bit whenever the function yields.

This is the fundamental idea of Motion Canvas. `yield` means: "The current frame is ready, display it on the screen and come back to me later."

With that in mind, we can make a circle flicker on the screen using the following code:

```tsx
export default makeScene2D(function* (view) {
  const circle = createRef<Circle>();
  view.add(<Circle ref={circle} width={100} height={100} />);

  circle().fill('red');
  yield;
  circle().fill('blue');
  yield;
  circle().fill('red');
  yield;
});
```

Needless to say, it would be extremely cumbersome if we had to write all animations like that. Fortunately, JavaScript has another keyword for use within generators - `yield*`. It allows us to delegate the yielding to another generator.

For instance, we could extract the flickering code from the above example to a separate generator and delegate our scene function to it:

```tsx
import {ThreadGenerator} from '@revideo/core';

export default makeScene2D(function* (view) {
  const circle = createRef<Circle>();
  view.add(<Circle ref={circle} width={100} height={100} />);
```

```
  yield* flicker(circle());
});

function* flicker(circle: Circle): ThreadGenerator {
  circle.fill('red');
  yield;
  circle.fill('blue');
  yield;
  circle.fill('red');
  yield;
}
```

The resulting animation is exactly the same, but now we have a reusable function that we can use whenever we need some flickering.

Motion Canvas provides a lot of useful generators like this. You may remember this snippet:

```ts
yield * myCircle().fill('#e6a700', 1);
```

It animates the fill color of the circle from its current value to `#e6a700` over a span of one second. As you may guess, the result of calling `fill('#e6a700', 1)` is another generator to which we can redirect our scene function. Generators like this are called tweens, because they animate be**tween** two values. You can read more about them in the [tweening](/tweening) section.

Flow Generators

Another kind of generators are _flow generators_. They take one or more generators as their input and combine them together. We've mentioned the `all()` generator in the quickstart section, there's a few more:

`all`

<ApiSnippet url={'/api/core/flow#all'} />
<hr />

`any`

<ApiSnippet url={'/api/core/flow#any'} />
<hr />

`chain`

<ApiSnippet url={'/api/core/flow#chain'} />
<hr />

`delay`

<ApiSnippet url={'/api/core/flow#delay'} />
<hr />

`sequence`

<ApiSnippet url={'/api/core/flow#sequence'} />
<hr />

`loop`

<ApiSnippet url={'/api/core/flow#loop'} />
<hr />

Looping

There are many ways to animate multiple objects. Here are some examples. Try using them in the below editor.

```tsx editor ratio=2
import {makeScene2D, Rect} from '@revideo/2d';
import {all, waitFor, makeRef, range} from '@revideo/core';
```

```
export default makeScene2D(function* (view) {
  const rects: Rect[] = [];

  // Create some rects
  view.add(
    range(5).map(i => (
      <Rect
        ref={makeRef(rects, i)}
        width={100}
        height={100}
        x={-250 + 125 * i}
        fill="#88C0D0"
        radius={10}
      />
    )),
  );

  yield* waitFor(1);

  // Animate them
  yield* all(
    ...rects.map(rect => rect.position.y(100, 1).to(-100, 2).to(0, 1)),
  );
});
```

Using `Array.map` and `all`

This is one of the most elegant ways to do simple tweens, but requires nesting `all` to do multiple tweens on an object since the `map` callback must return a `ThreadGenerator`.

```tsx
yield *
  all(
    ...rects.map(rect =>
```

```
    // No yield or anything; we return this generator and deal with it outside
    rect.position.y(100, 1).to(-100, 2).to(0, 1),
  ),
);
```

Using a `for` loop and `all`

This is similar to above, but uses a `for` loop and an array of generators.

```tsx
const generators = [];
for (const rect of rects) {
  // No yield here, just store the generators.
  generators.push(rect.position.y(100, 1).to(-100, 2).to(0, 1));
}

// Run all of the generators.
yield * all(...generators);
```

Using a `for` loop

This is a bit of a cumbersome option because you have to figure out how long it would take for the generator in the loop to complete, but is useful in some situations.

```tsx
for (const rect of rects) {
  // Note the absence of a * after this yield
  yield rect.position.y(100, 1).to(-100, 2).to(0, 1);
}

// Wait for the duration of the above generators
yield * waitFor(4);
```

File Path: revideo-docs/motion-canvas/hierarchy.mdx
---
sidebar_position: 3
slug: /hierarchy
---

import Mermaid from '@theme/Mermaid';
import CodeBlock from '@theme/CodeBlock';
import ApiSnippet from '@site/src/components/Api/ApiSnippet';

Scene hierarchy

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Scenes are collections of nodes displayed in your animation. They're organized in a tree hierarchy, with the scene view at its root. This concept is similar to the Document Object Model used to represent HTML and XML documents.

Here's an example of a simple scene hierarchy together with its object representation:

```tsx
view.add(
 <>
  <Circle />
  <Layout>
   <Rect />
   <Txt>Hi</Txt>
  </Layout>
 </>,
);
```

```mermaid
graph TD;
 view[Scene View]
 circle([Circle])
 layout([Layout])
 rect([Rect])
 text([text 'Hi'])
  view-->circle;
  view-->layout;
  layout-->rect;
  layout-->text;
```

Each node is an instance of a class extending the base [`Node`][node] class. To make the code more readable, Motion Canvas uses a custom [JSX](https://reactjs.org/docs/introducing-jsx.html) runtime. This way, instead of instantiating the nodes ourselves, we can write an XML-like markup. Note that Motion Canvas does **not** use React itself, only JSX. There's no virtual DOM or reconciliation and the JSX tags are mapped directly to Node instances. These two code snippets are equivalent:

```tsx
// JSX
view.add(
 <>
  <Circle />
  <Layout>
   <Rect />
```

```
    <Txt>Hi</Txt>
   </Layout>
 </>,
);
```

```tsx
// No JSX
view.add([
 new Circle({}),
 new Layout({
  children: [
   // highlight-start
   new Rect({}),
   new Txt({text: 'Hi'}),
  ],
 }),
]);
```

Modifying the hierarchy

After the hierarchy has been created, it's still possible to add, remove, and rearrange nodes at any time. The [`Node`][node] class contains the [`children`](/api/2d/components/Node#children) and [`parent`](/api/2d/components/Node#parent) properties that can be used to traverse the tree. But in order to modify it, it's recommended to use the following helper methods:

`Node.add`

<ApiSnippet url={'/api/2d/components/Node#add'} />
<hr />

`Node.insert`

<ApiSnippet url={'/api/2d/components/Node#insert'} />
<hr />

`Node.remove`

<ApiSnippet url={'/api/2d/components/Node#remove'} />
<hr />

`Node.reparent`

<ApiSnippet url={'/api/2d/components/Node#reparent'} />
<hr />

`Node.moveUp`

<ApiSnippet url={'/api/2d/components/Node#moveUp'} />
<hr />

`Node.moveDown`

<ApiSnippet url={'/api/2d/components/Node#moveDown'} />
<hr />

`Node.moveToTop`

<ApiSnippet url={'/api/2d/components/Node#moveToTop'} />
<hr />

`Node.moveToBottom`

<ApiSnippet url={'/api/2d/components/Node#moveToBottom'} />
<hr />

`Node.moveTo`

<ApiSnippet url={'/api/2d/components/Node#moveTo'} />
<hr />

`Node.moveAbove`

<ApiSnippet url={'/api/2d/components/Node#moveAbove'} />
<hr />

`Node.moveBelow`

<ApiSnippet url={'/api/2d/components/Node#moveBelow'} />
<hr />

`Node.removeChildren`

<ApiSnippet url={'/api/2d/components/Node#removeChildren'} />
<hr />

Querying the hierarchy

Sometimes it can be useful to traverse the hierarchy and find some specific nodes. In this documentation, we'll be referring to this process as _querying_. Consider the following animation:

```tsx editor
import {makeScene2D, Layout, Txt, Circle, Rect, is} from '@revideo/2d';
import {all} from '@revideo/core';

export default makeScene2D(function* (view) {
  view.add(
    <Layout layout gap={20} alignItems={'center'}>
      <Txt fill={'white'}>Example</Txt>
      <Rect fill={'#f3303f'} padding={20} gap={20}>
        <Txt fill={'white'}>42</Txt>
```

```
      <Circle size={60} fill={'#FFC66D'} />
      <Txt fill={'white'}>!!!</Txt>
    </Rect>
  </Layout>,
 );

 const texts = view.findAll(is(Txt));

 yield* all(...texts.map(text => text.fill('#FFC66D', 1).back(1)));
});
```

It contains multiple text nodes whose color oscillates between white and yellow. To achieve that, we used `view.findAll(is(Txt))` to search through all descendants of the view node and select only those of type `Txt`. The first argument passed to the [`findAll`](/api/2d/components/Node#findAll) method is a
so-called `predicate`. It's a function that takes a node and returns `true` if it's a node we're looking for.

For instance, if we wanted to find all nodes whose scale x is greater than `1`, we could write:

```ts
const wideNodes = view.findAll(node => node.scale.x() > 1);
```

Knowing this, we could try to find all nodes of type `Txt` as follows:

```ts
const texts = view.findAll(node => node instanceof Txt);
```

But Motion Canvas comes with a helpful utility function called [`is`](/api/2d/utils#is) that can create this predicate for us:

```ts
```

```
import {is} from '@revideo/2d';
// ...
const texts = view.findAll(is(Txt));
```

These can be used with any JavaScript function that accepts a predicate. The `findAll` method has been implemented to traverse all descendants of a node, but
if we wanted to query only the direct children, we could retrieve the [`children`](/api/2d/components/Node#children) array and call the built-in `filter` method with our predicate:

```ts
const textChildren = someParent.children().filter(is(Txt));
```

There are a few other methods that can be used to query the hierarchy depending on your needs:

`Node.findAll`

<ApiSnippet url={'/api/2d/components/Node#findAll'} />
<hr />

`Node.findFirst`

<ApiSnippet url={'/api/2d/components/Node#findFirst'} />
<hr />

`Node.findLast`

<ApiSnippet url={'/api/2d/components/Node#findLast'} />
<hr />

`Node.findAncestor`

<ApiSnippet url={'/api/2d/components/Node#findAncestor'} />

<hr />

[node]: /api/2d/components/Node

File Path: revideo-docs/motion-canvas/layouts.mdx
---
sidebar_position: 5
slug: /layouts
---

import AnimationPlayer from '@site/src/components/AnimationPlayer';
import ApiSnippet from '@site/src/components/Api/ApiSnippet';

Layouts

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

<AnimationPlayer name="layout" banner />

Layouts allow you to arrange your nodes using [Flexbox][flexbox]. Any node extending the [`Layout`](/api/2d/components/Layout) node can become a part of the layout. This includes, but is not limited to:
[`Rect`](/api/2d/components/Rect), [`Circle`](/api/2d/components/Circle), and [`Img`](/api/2d/components/Img).

Layout root

Layouts are an opt-in feature, meaning that they need to be enabled. It's done by setting the [`layout`](/api/2d/components/Layout#layout) property on the Node
that we want to become the root of our layout:

```tsx
// ↓ layout root
<Rect layout>
  {/* ↓ layout child */}
  <Circle width={320} height={320} />
</Rect>
```

In the example above, we marked the `<Rect>` as the layout root. This will cause the position and size of its descendants to be controlled by Flexbox (In this case there's only one valid descendant: `<Circle>`). The layout root itself is treated differently than its children - its size is controlled by Flexbox, but the position stays unaffected.

:::info

Just setting the `layout` property doesn't always turn the node into a layout root. If the node is already a part of the layout, it will be treated like the rest of the descendants:

```tsx
// ↓ layout root
<Rect layout>
 {/* ↓ layout child, NOT a layout root */}
 <Rect layout>
  {/* ↓ layout child */}
  <Circle width={320} height={320} />
 </Rect>
</Rect>
```

:::

Size and offset

Aside from the position, rotation, and scale, any node extending the `Layout` class has additional `size` and `offset` properties:

`Layout.size`

<ApiSnippet url={'/api/2d/components/Layout#size'} />
<hr />

`Layout.offset`

<ApiSnippet url={'/api/2d/components/Layout#offset'} />
<hr />

Cardinal directions

Layout nodes come with a set of helper properties that let you position them in respect to their edges/corners. In the example below we use them to place two squares on the left and right side of a gray rectangle. The yellow square is positioned so that its right edge is in the same place as the left edge of the rectangle. Meanwhile, the red square is placed so that its bottom left corner aligns with the bottom right corner of the rectangle. All possible directions include: [`middle`](/api/2d/components/Layout#middle),
[`top`](/api/2d/components/Layout#top),
[`bottom`](/api/2d/components/Layout#bottom),
[`left`](/api/2d/components/Layout#left),
[`right`](/api/2d/components/Layout#right),
[`topLeft`](/api/2d/components/Layout#topLeft),
[`topRight`](/api/2d/components/Layout#topRight),
[`bottomLeft`](/api/2d/components/Layout#bottomLeft), and
[`bottomRight`](/api/2d/components/Layout#bottomRight).

```tsx editor
import {makeScene2D, Rect} from '@revideo/2d';
import {createRef} from '@revideo/core';

export default makeScene2D(function* (view) {
  const rect = createRef<Rect>();

  view.add(
    <>
      <Rect
        ref={rect}
        width={200}
        height={100}
        rotation={-10}
        fill={'#333333'}
      />
```

```
    <Rect
      size={50}
      fill={'#e6a700'}
      rotation={rect().rotation}
      // Try changing "right" to "top"
      right={rect().left}
    />
    <Rect
      size={100}
      fill={'#e13238'}
      rotation={10}
      bottomLeft={rect().bottomRight}
    />
  </>,
 );

 yield* rect().rotation(10, 1).to(-10, 1);
});
```

Flexbox configuration

Most flexbox attributes available in CSS are available as [`Layout` properties](/api/2d/components/Layout#Properties). You can check out this [Flexbox guide][flexbox] to better understand how they work. The most useful properties are listed below:

`Layout.padding`

<ApiSnippet url={'/api/2d/components/Layout#padding'} />
<hr />

`Layout.margin`

<ApiSnippet url={'/api/2d/components/Layout#margin'} />
<hr />

`Layout.gap`

<ApiSnippet url={'/api/2d/components/Layout#gap'} />
<hr />

`Layout.direction`

<ApiSnippet url={'/api/2d/components/Layout#direction'} />
<hr />

`Layout.alignItems`

<ApiSnippet url={'/api/2d/components/Layout#alignItems'} />
<hr />

`Layout.justifyContent`

<ApiSnippet url={'/api/2d/components/Layout#justifyContent'} />
<hr />

Groups

Nodes that don't extend the `Layout` class, such as the `Node` itself, are unaffected by the layout and are treated as if they were never there. This lets you apply filters and transformations to layout nodes without affecting the hierarchy.

From the layout's perspective, all `<Rect>`s in the example below are siblings:

```tsx
<Layout direction={'column'} width={960} gap={40} layout>
 <Node opacity={0.1}>
  <Rect height={240} fill={'#ff6470'} />
  <Rect height={240} fill={'#ff6470'} />
 </Node>
 <Rect height={240} fill={'#ff6470'} />
</Layout>
```

```

<AnimationPlayer name="layout-group" small />

[flexbox]: https://css-tricks.com/snippets/css/a-guide-to-flexbox/
```

File Path: revideo-docs/motion-canvas/logging.mdx
---
sidebar_position: 12
slug: /logging

Logging

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

One method of debugging your code or animation flow is using logging messages. For this, revideo has its own built-in way to log messages.

To get a reference to the Logger in revideo you can use the `useLogger` function:

```tsx
import {makeScene2D} from '@revideo/2d';
import {useLogger} from '@revideo/core';

export default makeScene2D(function* (view) {
  const logger = useLogger();
  // ...
});
```

Basic

Now that we know how to get a reference to the `Logger` we can take a look at different ways to log messages. On way is to use standard logging functions like `debug`, `info`, `warn` and `error` and simply log a string:

```tsx
logger.debug('Just here to debug some code.');
logger.info('All fine just a little info.');
logger.warn('Be careful something has gone wrong.');
```

logger.error('Ups. An error occured.');
```

These messages get then displayed in the UI under the `Console` tab on the left side.

## Payloads

In some cases you might want to have a bit more detail in your log messages like a `stacktrace` or an `object`. You can use payloads to provide more information to your log messages.

```ts
logger.debug({
  message: 'Some more advanced logging',
  remarks: 'Some remarks about this log. Can also contain <b>HTML</b> tags.',
  object: {
    someProperty: 'some property value',
  },
  durationMs: 200,
  stack: new Error('').stack,
});
```

This creates a collapsed log message in the UI which can be expanded to view all the details provided.

:::tip

If you quickly want to debug something you can also `debug()`. That way you don't have to `useLogger` manually and create a payload.

:::

Profiling
```

Besides logging messages its also possible to profile certain sections of code with the `Logger`:

```ts
logger.profile('id'); // <-- starts the profiling
// some expensive calculation
logger.profile('id'); // <-- ends the profiling
```

File Path: revideo-docs/motion-canvas/positioning.mdx
---
sidebar_position: 4
slug: /positioning
---

import AnimationPlayer from '@site/src/components/AnimationPlayer';
import ApiSnippet from '@site/src/components/Api/ApiSnippet';

Positioning

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

<AnimationPlayer name="positioning" banner />

Motion Canvas uses a Cartesian coordinate system. Its origin is located in the center of the scene, with the x-axis going to the right and the y-axis going down.

Transform

All nodes are positioned relative to their parents. This means that any transformations applied to the parent are also applied to its children. The transform of each node consists of the following properties:

`Node.position`

<ApiSnippet url={'/api/2d/components/Node#position'} />
<hr />

`Node.scale`

<ApiSnippet url={'/api/2d/components/Node#scale'} />
<hr />

`Node.rotation`

<ApiSnippet url={'/api/2d/components/Node#rotation'} />
<hr />

Absolute transform

Each of the basic transform properties has a dedicated helper method that operates in world space.

This can be helpful, for instance, when we need to match the transforms of two nodes located within different parents. Consider the following example:

```tsx
const circleA = createRef<Node>();
const circleB = createRef<Node>();

view.add(
 <>
  <Node position={[200, 100]}>
   <Circle
    position={[0, 100]}
    ref={circleA}
    width={20}
    height={20}
    fill={'white'}
   />
  </Node>
  <Circle ref={circleB} width={10} height={10} fill={'red'} />
 </>,
);

circleB().absolutePosition(circleA().absolutePosition());
```

We access the absolute position (position in world space) of `circleA` and assign it as the absolute position of `circleB`. This will move the `circleB` right on top of `circleA`.

:::info

Note that we still need to set the `absolutePosition` of `circleB` and not just the `position`. It may seem redundant since `circleB` is a direct child of the scene view. But the local space of the scene view is **not** the same as the world space.

:::

All available world-space properties are listed below:

`Node.absolutePosition`

<ApiSnippet url={'/api/2d/components/Node#absolutePosition'} />
<hr />

`Node.absoluteScale`

<ApiSnippet url={'/api/2d/components/Node#absoluteScale'} />
<hr />

`Node.absoluteRotation`

<ApiSnippet url={'/api/2d/components/Node#absoluteRotation'} />
<hr />

Matrices

For more advanced uses, nodes expose all the matrices necessary to map vectors from one space to another. For example, the helper properties described above could be reimplemented using the `worldToParent` and `localToWorld` matrices:

```ts
// getting the absolute position:
node.absolutePosition();
```

```
// same as:
Vector2.zero.transformAsPoint(node.localToWorld());

// setting the absolute position:
node.absolutePosition(vector);
// same as:
node.position(vector.transformAsPoint(node.worldToParent()));
```

The available matrices include:

`Node.localToWorld`

<ApiSnippet url={'/api/2d/components/Node#localToWorld'} />
<hr />

`Node.worldToLocal`

<ApiSnippet url={'/api/2d/components/Node#worldToLocal'} />
<hr />

`Node.localToParent`

<ApiSnippet url={'/api/2d/components/Node#localToParent'} />
<hr />

`Node.worldToParent`

<ApiSnippet url={'/api/2d/components/Node#worldToParent'} />
<hr />

File Path: revideo-docs/motion-canvas/project-variables.mdx
---
sidebar_position: 22
slug: /project-variables

Project variables

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Making animations dynamic can be achieved using project variables. These could be used for light/dark themes, changing Node styling or editing text content. Project variables make use of signals - allowing variables to be updated during animations.

Adding variables via the player component can be done by passing a stringified json object to the variables attribute:

```html
<revideo-player
  src="/path/to/project.js"
  variables='{"circleFill":"red"}'
></revideo-player>
```

They can also be added using makeProject():

```ts
export default makeProject({
  scenes: [example],
  variables: {circleFill: 'red'},
});
```

Accessing the variables inside of a scene is through
[`useScene()`](/api/core/utils#useScene):

```ts
const circleFill = useScene().variables.get('circleFill', 'blue');
```

File Path: revideo-docs/motion-canvas/random.mdx
---
sidebar_position: 24
slug: /random-values
---

import AnimationPlayer from '@site/src/components/AnimationPlayer';
import CodeBlock from '@theme/CodeBlock';
import source from '!!raw-loader!@revideo/examples/src/scenes/random';

Random values

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

<AnimationPlayer name="random" banner />

Randomly generated values can be used to create a sense of variety and
unpredictability in your animation. In Motion Canvas, it's achieved using the
[`useRandom()`](/api/core/utils#useRandom) function. It returns a random
number
generator (RNG) for the current scene:

```ts
import {useRandom} from '@revideo/core';

const random = useRandom();
const integer = random.nextInt(0, 10);
```

In this case, calling `nextInt()` will return an integer in the range from 0 to
10 (exclusive). Check the [`Random` api](/api/core/scenes/Random) to see all
available methods.

Unlike `Math.random()`, `useRandom()` is completely reproducible - each time
the
animation is played the generated values will be exactly the same. The seed used

to generate these numbers is stored in the meta file of each scene.

You can also provide your own seed to find a sequence of numbers that best suits your needs:

```ts
const random = useRandom(123);
```

The animation at the top of this page uses a random number generator to vary the height of rectangles and make them look like a sound-wave:

<CodeBlock language="tsx">{source}</CodeBlock>

File Path: revideo-docs/motion-canvas/references.mdx
---
sidebar_position: 7
slug: /references

References

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Usually, when creating a node, we want to store a reference to it, so we can animate it later. One way to do that is by assigning it to a variable first, and then adding it to the scene:

```tsx
const circle = <Circle />;
view.add(circle);

// we can now animate our circle:
yield * circle.scale(2, 0.3);
```

:::info

If you're used to libraries such as React, the above example may seem strange. In Motion Canvas, the JSX components immediately create and return an instance of the given class. It's completely valid to store it as a reference and use it throughout the animation.

:::

But this approach doesn't scale well. The more nodes we add, the harder it gets to see the overall structure of our scene. Consider the following example:

```tsx
const rectA = <Rect />;
```

```
const rectB = <Rect />;
const circle = <Circle>{rectA}</Circle>;
view.add(
  <Layout>
    {circle}
    {rectB}
  </Layout>,
);
```

And now compare it to a version that doesn't store any references:

```tsx
view.add(
  <Layout>
    <Circle>
      <Rect />
    </Circle>
    <Rect />
  </Layout>,
);
```

If you find the latter example more readable, this guide is for you.

`ref` property

Each node in Motion Canvas has a property called `ref` that allows you to create a reference to said node. It accepts a callback that will be invoked right after the node has been created, with the first argument being the newly created instance.

With this in mind, we can rewrite the initial example as:

```tsx
let circle: Circle;
view.add(
```

```
  <Circle
    ref={instance => {
      circle = instance;
    }}
  />,
);

yield * circle.scale(2, 0.3);
```

Using the `ref` property in this way is not really practical, and we wouldn't recommend it. But it's crucial to understand how it works because all the upcoming methods use this property as a base.

`createRef()` function

The preferred way of using the `ref` property is in conjunction with the [`createRef()`](/api/core/utils#createRef) function. Continuing with our example, we can rewrite it as:

```tsx
import {createRef} from '@revideo/core';

// ...

const circle = createRef<Circle>();
view.add(<Circle ref={circle} />);

yield * circle().scale(2, 0.3);
```

Notice that `circle` is no longer just a variable that points to our circle. Instead, it's a [signal-like](/signals) function that can be used to access it. Invoking it without any arguments (`circle()`) returns our instance.

Going back to the example with the more complex scene, we can now rewrite it as:

```tsx
const rectA = createRef<Rect>();
const rectB = createRef<Rect>();
const circle = createRef<Circle>();
view.add(
  <Layout>
    <Circle ref={circle}>
      <Rect ref={rectA} />
    </Circle>
    <Rect ref={rectB} />
  </Layout>,
);
```

`makeRef()` function

Another common use case of the `ref` property is to assign the newly created instance to a property of some object. In the following example, we assign our circle to `circle.instance` (We'll talk about why this may be useful in a bit):

```tsx
const circle = {instance: null as Circle};
view.add(
  <Circle
    // highlight-start
    ref={instance => {
      circle.instance = instance;
    }}
    // highlight-end
  />,
);
```

We can use the [`makeRef()`](/api/core/utils#makeRef) function to simplify this process:

```tsx
```

```
import {makeRef} from '@revideo/core';

// ...

const circle = {instance: null as Circle};
view.add(
  // highlight-next-line
  <Circle ref={makeRef(circle, 'instance')} />,
);
```

Array of references

`makeRef()` can be particularly useful when we create an array of nodes and want to grab references to all of them:

```tsx
const circles: Circle[] = [];
view.add(
  <Layout>
    {range(10).map(index => (
      // highlight-next-line
      <Circle ref={makeRef(circles, index)} />
    ))}
  </Layout>,
);
```

In JavaScript, arrays are objects whose properties are their indices. So `makeRef(circles, index)` will set the nth element of our array to the created circle. As a result, we end up with an array of size `10` filled with circles that we can use to animate all of them.

You can also use the [`createRefArray()`](/api/core/utils#createRefArray) helper function to achieve the same result:

```tsx
```

```
import {createRefArray, range} from '@revideo/core';

// ...

const circles = createRefArray<Circle>();
view.add(
  <Layout>
    {range(10).map(() => (
      <Circle ref={circles} />
    ))}
  </Layout>,
);
```

This time we don't specify the index. Whenever we pass the `circles` array to the `ref` property, the newly created circle will be appended to our array.

:::tip

Check out [the looping section in the flow guide](/flow#looping) to see how an array of references can be used to orchestrate animations.

:::

Custom functions

`makeRef()` can also be used to return more than one reference from a custom function component:

```tsx
function Label({
  refs,
  children,
}: {
  refs: {rect: Rect; text: Txt};
  children: string;
}) {
```

```
  return (
    // highlight-next-line
    <Rect ref={makeRef(refs, 'rect')}>
      // highlight-next-line
      <Txt ref={makeRef(refs, 'text')}>{children}</Txt>
    </Rect>
  );
}

const label = {rect: null as Rect, text: null as Txt};
view.add(<Label refs={label}>HELLO</Label>);

// we can now animate both the Rect and the Text of our label:
yield * label.rect.opacity(2, 0.3);
yield * label.text.fontSize(24, 0.3);
```

In this example, we define a function component called `Label` consisting of a rectangle with some text inside. When using the component, we use the `refs` property to pass the `label` object created by us. `makeRef()` is then used to fill this object with all the necessary references.

`createRefMap()` function

As the scene grows in complexity, declaring a reference for each node can become tedious. The [`createRefMap()`](/api/core/utils#createRefMap) helper function lets us group references together based on the type of the node:

```tsx
import {createRefMap} from '@revideo/core';

// ...

const labels = createRefMap<Txt>();
view.add(
  <>
    <Txt ref={labels.a}>A</Txt>
```

```
  <Txt ref={labels.b}>B</Txt>
  <Txt ref={labels.c}>C</Txt>
 </>,
);
```

The returned object is a map that can store however many references we need. In the above example, we assign three `Txt` references under the keys `a`, `b`, and `c`. Simply accessing a property of the map, like `labels.a` will create a reference for us. The names of the properties are arbitrary and can be anything we want.

Later on, we can retrieve the references using the same keys:

```tsx
yield * labels.a().text('A changes', 0.3);
yield * labels.b().text('B changes', 0.3);
yield * labels.c().text('C changes', 0.3);
```

To check if a reference exists, we can use the `in` operator. This will avoid creating a reference:

```tsx
if ('d' in labels) {
  yield * labels.d().text('D changes', 0.3);
}
```

The returned object comes with a `mapRefs` method that lets us map over all references in the map. It's similar to the [Array.prototype.map][array map] function:

```ts
yield * all(...labels.mapRefs(label => label.fill('white', 0.3)));
```

`makeRefs()` function

Looking at the previous example, you may notice that we had to define the `refs` type twice. First in the `Label` declaration and then again when creating the `label` object:

```tsx
function Label({
  refs,
  children,
}: {
  // highlight-next-line
  refs: {rect: Rect; text: Txt};
  children: string;
}) {
  return (
    <Rect ref={makeRef(refs, 'rect')}>
      <Txt ref={makeRef(refs, 'text')}>{children}</Txt>
    </Rect>
  );
}

// highlight-next-line
const label = {rect: null as Rect, text: null as Txt};
view.add(<Label refs={label}>HELLO</Label>);
```

We can use [`makeRefs()`](/api/core/utils#makeRefs) to eliminate this redundancy. It can extract the type from the `Label` declaration and create an empty object matching it:

```tsx
import {makeRef, makeRefs} from '@revideo/core';

// ...

function Label({
```

```
  refs,
  children,
}: {
  // highlight-next-line
  refs: {rect: Rect; text: Txt};
  children: string;
}) {
  return (
    <Rect ref={makeRef(refs, 'rect')}>
      <Txt ref={makeRef(refs, 'text')}>{children}</Txt>
    </Rect>
  );
}

// highlight-next-line
const label = makeRefs<typeof Label>();
view.add(<Label refs={label}>HELLO</Label>);
```

[array map]:
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

File Path: revideo-docs/motion-canvas/shaders.mdx
---
sidebar_position: 25
slug: /shaders
---

import ExperimentalWarning from '@site/src/components/ExperimentalWarning';

Shaders

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Shaders allow you to apply custom effects to any node using WebGL.

<ExperimentalWarning />

Shaders can be specified using the [`shaders`](/api/2d/components/
Node#shaders)
property. In the simplest case, the value should be a string containing the GLSL
code for the fragment shader:

```tsx
// highlight-next-line
import myShader from './myShader.glsl';

//...

view.add(
  <Circle
    size={200}
    fill="lightseagreen"
    // highlight-next-line
    shaders={myShader}
  />,
);
```

Below is an example of a simple shader that inverts the colors of the node:

```glsl title="myShader.glsl"
#version 300 es
precision highp float;

#include "@revideo/core/shaders/common.glsl"

void main() {
  outColor = texture(sourceTexture, sourceUV);
  outColor.rgb = 1.0 - outColor.rgb;
}
```

GLSL Preprocessor

Motion Canvas comes with a simple GLSL preprocessor that lets you include files using the `#include` directive:

```glsl
#include "path-to-file"
```

The path is resolved using the same rules as `import` statements in JavaScript. It can point to a relative file:

```glsl
#include "../utils/math.glsl"
```

Or to a file from another package:

```glsl
#include "@revideo/core/shaders/common.glsl"
```

For convenience, a GLSL file can be imported only once per shader. Each subsequent import of the same file will be ignored so `#ifdef` guards are not necessary.

Default uniforms

The following uniforms are available in all shaders:

```glsl
in vec2 screenUV;
in vec2 sourceUV;
in vec2 destinationUV;

out vec4 outColor;

uniform float time;
uniform float deltaTime;
uniform float framerate;
uniform int frame;
uniform vec2 resolution;
uniform sampler2D sourceTexture;
uniform sampler2D destinationTexture;
uniform mat4 sourceMatrix;
uniform mat4 destinationMatrix;
```

They can be included using the following directive:

```glsl
#include "@revideo/core/shaders/common.glsl"
```

Source and Destination

Shaders in Motion Canvas follow the same idea as [`globalCompositeOperation`][globalCompositeOperation] in 2D canvas. The `sourceTexture` contains the node being rendered, and the `destinationTexture`

contains what has already been rendered to the screen. These two can be sampled using `sourceUV` and `destinationUV` respectively, and then combined in various
ways to produce the desired result.

Custom uniforms

You can pass custom uniforms to the shader by replacing the shader string with a configuration object:

```tsx
// highlight-next-line
import myShader from './myShader.glsl';

//...

view.add(
 <Circle
  size={200}
  fill="lightseagreen"
  // highlight-next-line
  shaders={{
   fragment: myShader,
   uniforms: {
    myFloat: 0.5,
    myVec2: new Vector2(2, 5),
    myColor: new Color('blue'),
   },
  }}
 />,
);
```

The `uniforms` property is an object where the keys are the names of the uniforms and the values are what's passed to the shader.

The type of the uniform is inferred from the value:

TypeScript	GLSL
`number`	`float`
`[number, number]`	`vec2`
`[number, number, number]`	`vec3`
`[number, number, number, number]`	`vec4`
[`Color`][Color]	`vec4`
[`Vector2`][Vector2]	`vec2`
[`BBox`][BBox]	`vec4`
[`Spacing`][Spacing]	`vec4`

With that in mind, the uniforms from the above example will be available in the shader as:

```glsl title="myShader.glsl"
uniform float myFloat;
uniform vec2 myVec2;
uniform vec4 myColor;
```

It's also possible to create custom classes that can be passed as uniforms by implementing the [`WebGLConvertible`][WebGLConvertible] interface.

Caching

When a node is cached, its contents are first rendered to a separate canvas and then transferred to the screen (You can read more about it in the [Filters and Effects](/filters-and-effects#cached-nodes) section) When a shader is applied to a descendant of a cached node, the `destinationTexture` will only contain the things drawn in the context of that cached node and nothing else. This is analogous to how composite operations work.

Any node with a shader is automatically cached - this lets us figure out the contents of the `sourceTexture` before shaders are run. Caching requires us to know the size and position of everything rendered by the node. This goes beyond its logical size. Things like shadows, strokes, and filters can make the

rendered area larger. We account for that in the case of built-in effects, but
for custom shaders you may need to adjust the cache size manually. The
[`cachePadding`](/api/2d/components/Node#cachePadding) property can be used
to
do exactly that. It specifies the extra space around the node that should be
included in the cache.

[globalCompositeOperation]:
https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/
globalCompositeOperation
[Color]: https://gka.github.io/chroma.js/#color
[Vector2]: /api/core/types/Vector2
[BBox]: /api/core/types/BBox
[Spacing]: /api/core/types/Spacing
[WebGLConvertible]: /api/core/types/WebGLConvertible

File Path: revideo-docs/motion-canvas/signals.mdx
---
sidebar_position: 6
slug: /signals
---

import AnimationPlayer from '@site/src/components/AnimationPlayer';
import CodeBlock from '@theme/CodeBlock';
import signalSource from '!!raw-loader!@revideo/examples/src/scenes/node-signal';

Signals

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Signals represent a value that may change over time. They can be used to define dependencies between the state of the animation. This way, when a value changes,
all other values that depend on it get automatically updated.

Overview

Signals for primitive types are created using the
[`createSignal()`](/api/core/signals#createSignal) function, where the first argument specifies their initial value:

```ts
import {createSignal} from '@revideo/core';

const signal = createSignal(0);
```

Additionally, each complex type has a static `createSignal()` method that can be used to create a signal for said type:

```ts
```

```
import {Vector2} from '@revideo/core';

const signal = Vector2.createSignal(Vector2.up);
```

Properties of every node are also represented by signals:

```tsx
const circle = <Circle />;

const signal = circle.fill;
```

Once created, signals can be invoked to perform one of the three possible actions (The action is chosen based on the number of arguments):

1. retrieve the value:
   ```ts
   const value = signal();
   ```
2. update the value:
   ```ts
   signal(3);
   ```
3. create a [tween](/tweening) for the value:
   ```ts
   yield * signal(2, 0.3);
   ```

Instead of the actual value, a signal can be provided with a function that computes the value dynamically. Consider the following example:

```ts
const radius = createSignal(1);
const area = createSignal(() => Math.PI * radius() * radius());

console.log(area()); // 3.141592653589793
```

```
radius(2);
console.log(area()); // 12.566370614359172
```

Here, the `area` signal uses the `radius` signal to compute its value.

Explanation

To better understand how signals work, let's modify the example from before to see when exactly the area is calculated:

```ts
const radius = createSignal(1);
const area = createSignal(() => {
  console.log('area recalculated!');
  return Math.PI * radius() * radius();
});

area(); // area recalculated!
area();
radius(2);
area(); // area recalculated!
radius(3);
radius(4);
area(); // area recalculated!
```

This demonstrates three important aspects of signals:

Laziness

Signals are only calculated when their value is requested. The first `"area recalculated!"` message is logged to console only after `area()` is called.

Caching

Once the signal is calculated, its value is saved and then returned during subsequent calls to `area()`. That's why nothing is logged to the console during the second call. This aspect of signals makes them perfect for caching computationally heavy operations. In fact, Motion Canvas uses signals internally to cache things such as matrices.

Dependency tracking

The `area` signal keeps track of other signals it depends on. When we change the `radius` signal, the `area` signal is notified about that. But it doesn't get recalculated immediately - laziness is still at play. We can modify the radius however many times we want, but the `area` will be recalculated only once its value is requested again by calling `area()`.

`DEFAULT` values

Signals keep track of the initial values specified during creation. At any time, we can reset a signal to its initial value by passing the [`DEFAULT`](/api/core/signals#DEFAULT) symbol to it:

```ts
import {DEFAULT, createSignal} from '@revideo/core';

const signal = createSignal(3); // <- initial value is 3
signal(2);
signal(); // <- value is now 2
signal(DEFAULT);
signal(); // <- value is reset back to 3
```

We can also use the [`DEFAULT`](/api/core/signals#DEFAULT) symbol for [tweening](/tweening):

```ts
yield * signal(DEFAULT, 2);
```

Resetting to the default value is especially useful with node properties. In the example below, we set the [`lineHeight`](/api/2d/components/Layout#lineHeight)
of the [`Txt`](/api/2d/components/Txt) node to `150%`. This will override its default value, which would be simply inherited from its parent:

```tsx
const text = createRef<Txt>();
view.add(
  <Txt lineHeight={'150%'} ref={text}>
    Hello world!
  </Txt>,
);
```

If we want to reset the [`lineHeight`](/api/2d/components/Layout#lineHeight) back to the default, inherited value, we can do so with
[`DEFAULT`](/api/core/signals#DEFAULT):

```ts
text().lineHeight(DEFAULT);
```

Complex example

We can use the fact that properties of nodes are represented by signals to construct scenes that automatically update when the data changes. Following the previous example, let's create a visualisation for the area of the circle:

<AnimationPlayer small name="node-signal" />

Below you'll find the code used to create this animation. We highlighted all the places where signals are used:

<CodeBlock language="tsx">{signalSource}</CodeBlock>

With this setup, all we need to do is animate the `radius` signal, and the rest

of the scene will adjust accordingly:

```ts
yield * radius(4, 2).to(3, 2);
```

File Path: revideo-docs/motion-canvas/spawners.mdx
---
sidebar_position: 18
slug: /spawners

Spawners

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Sometimes we want the children of a given node to be reactive. In other words, we want them to change according to some external state. Consider the following example:

```tsx
const count = createSignal(10);

view.add(
  <Layout layout>
    {range(count()).map(() => (
      <Circle size={32} fill={'white'} />
    ))}
  </Layout>,
);
```

We first create the `count` signal and then use its value to generate N number of circles.

This example is not reactive - changing the `count` signal won't change the number of circles inside the `Layout` node. We can fix that by using a function that returns the children instead of writing them directly:

```tsx
const count = createSignal(10);
```

```
view.add(
  <Layout layout>
    {() => range(count()).map(() => <Circle size={32} fill={'white'} />)}
  </Layout>,
);
```

Throughout this guide, we will refer to functions that return children as **spawners**. Like any other signal, this function will keep track of its dependencies and recompute its value whenever they change. We can animate our
`count` signal to see if it works:

```tsx editor
import {makeScene2D, Layout, Circle} from '@revideo/2d';
import {createSignal, linear, range} from '@revideo/core';

export default makeScene2D(function* (view) {
  const count = createSignal(10);

  view.add(
    <Layout layout>
      {() => range(count()).map(() => <Circle size={32} fill={'white'} />)}
    </Layout>,
  );

  yield* count(3, 2, linear).wait(1).back(2);
});
```

It's important to remember that creating new nodes comes with some overhead. If our spawner happens to generate a large number of nodes and its dependencies change every frame, it may drastically reduce the playback's performance. To counteract this, we can use an object pool that will let us reuse the same nodes instead of recreating them each time:

```tsx
```

```
const count = createSignal(10);

const pool = range(64).map(i => (
  <Circle x={i * 32} width={32} height={32} fill={'lightseagreen'} />
));

const layout = createRef<Layout>();
view.add(
  <Layout layout ref={layout}>
    {() => pool.slice(0, count())}
  </Layout>,
);
```

Apart from the spawner function, the pool should never be accessed directly. Use the helper methods on the parent object to get references to the spawned children:

```tsx
// ... continuing from above ...
let spawnedCircles = layout().childrenAs<Circle>();
yield * all(...spawnedCircles.map(circle => circle.scale(1.5, 1).to(1, 1)));
```

Be aware that the references returned by a call to `children()` may be invalidated when the number of spawned objects changes, and accessing the invalidated objects may cause undefined behavior. Try not to save references to spawned objects for too long, and use `children()` wherever possible to get the updated list of spawned objects.

File Path: revideo-docs/motion-canvas/time-events.mdx
---
sidebar_position: 9
slug: /time-events
---

import UI from '@site/src/components/UI';

Time Events

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

One of the pain points of creating explanatory animations using code is **synchronizing them with audio**. Imagine that you'd like a certain animation to start right after some specific verbal cue. With a code-only solution, you're left with hard-coding how long (or until which frame) you want to wait.

In Motion Canvas, it could look somewhat like this:

```ts
yield * animationOne();
yield * waitFor(3.1415); // but how long should we wait?
yield * animationTwo();
```

Evidently, this approach can get really tedious. Not only do you need to somehow find the exact timestamp to wait for, but also each time you modify the voiceover, you have to go through your code and adjust these hard-coded numbers.

That's why Motion Canvas allows you to edit these delays not through code, but through the editor. With the use of [`waitUntil`](/api/core/flow#waitUntil) you can pause the animation without specifying the actual duration:

```ts
yield * animationOne();
```

```
yield * waitUntil('event'); // wait for an event called "event"
yield * animationTwo();
```

This will cause the event to appear in the editor. From there, you can drag it to adjust its timing:

<UI className="image-inline" />

The dark trail behind the event illustrates its duration. It starts at the moment `waitUntil` was called - this is when the animation will pause. It will resume when the playhead reaches the event pill.

By default, adjusting a time event will also adjust all events that happen after it. This is useful when you want to extend or shorten a pause in your voiceover because correcting the first time event after the pause will also fix all events after it. You can hold <kbd>SHIFT</kbd> when editing an event to prevent this from happening.

Controlling animation duration

Aside from specifying _when_ something should happen, Time Events can also be used to control _how long_ something should last. You can use the [`useDuration`](/api/core/utils#useDuration) function to retrieve the duration of an event and use it in your animation:

```ts
yield * circle().scale(2, useDuration('event'));
```

File Path: revideo-docs/motion-canvas/transitions.mdx
---
sidebar_position: 10
slug: /transitions
---

import ApiSnippet from '@site/src/components/Api/ApiSnippet';
import AnimationPlayer from '@site/src/components/AnimationPlayer';

Transitions

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Transitions allow you to customize the way scenes transition from one into another. A transition is an animation performed at the beginning of the scene. It can modify the context of both the current and the previous scene.

Before we start

Make sure your project contains at least two scenes. In this example, we've prepared `firstScene.tsx` and `secondScene.tsx`, and configured our project to display one after the other. We'll be setting up our transitions in the second scene.

Make sure to put something different in both scenes to easier see the transitions.

```
my-animation/
└── src/
```

```
  ├── scenes/
  │   ├── firstScene.tsx
  │   └── secondScene.tsx
  └── project.ts
```

Pre-made transitions

Motion Canvas comes with a set of common transitions in a form of easy-to-use generators. To use them, `yield*` the transition generator at the beginning of the new scene:

```tsx title="src/scenes/secondScene.tsx"
export default makeScene2D(function* (view) {
 // set up the scene:
 view.add(/* your nodes here */);

 // perform a slide transition to the left:
 yield* slideTransition(Direction.Left);

 // proceed with the animation
 yield* waitFor(3);
});
```

<AnimationPlayer small name={'transitions'} link={'transitions-second'} />

:::caution

Make sure to add nodes to the view before yielding the transition generator. Otherwise, your scene will remain empty until the transition ends.

:::

All available transitions are listed below:

`slideTransition`

<ApiSnippet url={'/api/core/transitions#slideTransition'} />
<hr />

`zoomInTransition`

<ApiSnippet url={'/api/core/transitions#zoomInTransition'} />
<hr />

`zoomOutTransition`

<ApiSnippet url={'/api/core/transitions#zoomOutTransition'} />
<hr />

`fadeTransition`

<ApiSnippet url={'/api/core/transitions#fadeTransition'} />
<hr />

Custom transitions

You can use the [`useTransition`](/api/core/transitions#useTransition) function to implement custom transitions. It allows you to specify two callbacks that will modify the contexts of the current and previous scene respectively. The value it returns is a callback that you need to call once you finish the transition.

The transition template looks as follows:

```ts
// set up the transition
const endTransition = useTransition(
  currentContext => {
    // modify the context of the current scene
```

```
  },
  previousContext => {
    // modify the context of the previous scene
  },
);

// perform animations

// finish the transition
endTransition();
```

Here's how you could implement a simple slide transition:

```ts
export function* slideTransition(
  direction: Direction = Direction.Top,
  duration = 0.6,
): ThreadGenerator {
  const size = useScene().getSize();
  const position = size.getOriginOffset(direction).scale(2);
  const previousPosition = Vector2.createSignal();
  const currentPosition = Vector2.createSignal(position);

  // set up the transition
  const endTransition = useTransition(
    // modify the context of the current scene
    ctx => ctx.translate(currentPosition.x(), currentPosition.y()),
    // modify the context of the previous scene
    ctx => ctx.translate(previousPosition.x(), previousPosition.y()),
  );

  // perform animations
  yield* all(
    previousPosition(position.scale(-1), duration),
    currentPosition(Vector2.zero, duration),
  );
```

```
  // finish the transition
  endTransition();
}
```

Animate when transitioning

By default, Motion Canvas will transition to the next scene once the generator of the current scene has reached the end. In this case, the scene will freeze for the duration of the transition. You can use the [`finishScene`](/api/core/utils#finishScene) function to trigger the transition early, allowing the animation to continue while transitioning:

```tsx
export default makeScene2D(function* (view) {
  yield* animationOne();
  // trigger the transition early:
  finishScene();
  // continue animating:
  yield* animationTwo();
});
```

File Path: revideo-docs/motion-canvas/tweening.mdx
---
sidebar_position: 7
slug: /tweening
---

import AnimationPlayer from '@site/src/components/AnimationPlayer';
import CodeBlock from '@theme/CodeBlock';
import linearSource from '!!raw-loader!@revideo/examples/src/scenes/tweening-linear';
import saveRestoreSource from '!!raw-loader!@revideo/examples/src/scenes/tweening-save-restore';
import springSource from '!!raw-loader!@revideo/examples/src/scenes/tweening-spring';

Tweening

_Note: These docs were adopted from the original
[Motion Canvas](https://motioncanvas.io/docs/) docs_

Tweens are one of the fundamental building blocks of animation. They are a special type of generators that animate between two values over given time.

`tween` function

The simplest way to create a tween is via the
[`tween`](/api/core/tweening#tween) function:

<CodeBlock language="tsx">{linearSource}</CodeBlock>

In the example above, we animate the x coordinate of our circle from `-300` to `300` over a span of `2` second.

The [`tween`](/api/core/tweening#tween) function takes two parameters. The first
one specifies the tween duration in seconds. The second is a callback function that will be called each frame the tween takes place. The `value` parameter it

receives is a number ranging from `0` to `1`, informing us about the progress of the tween. We can use it to calculate the values that our tween animates. In the case of our circle, we use the `map` function to map the `value` range from `[0, 1]` to `[-300, 300]` and set it as the `x` coordinate:

<AnimationPlayer small name="tweening-linear" />

Timing functions

At the moment, our animation feels a bit unnatural. The speed with which the `value` parameter changes is constant, which in turn makes the circle move with constant speed. In real life, however, objects have inertia - they take time to speed up and slow down. We can simulate this behavior with [timing functions](/api/core/tweening).

A timing function takes a number in the range `[0, 1]` and returns another number in the same range but with a modified rate of change. Motion Canvas provides all [the most popular timing functions](https://easings.net/) (sometimes called easing functions) but since it's a normal JavaScript function you can create your own.

Let's use the [`easeInOutCubic`](/api/core/tweening#easeInOutCubic) function to fix our animation:

```ts
yield *
  tween(2, value => {
    circle().position.x(map(-300, 300, easeInOutCubic(value)));
  });
```

<AnimationPlayer small name="tweening-cubic" />

`easeInOut` means that the object will speed up at the start (`in`) and slow down at the end (`Out`). `Cubic` denotes the mathematical function used - in this case it's a cubic equation. Knowing that, a function called `easeOutQuad` would make the object start with full speed and then slow down at the end using

a quadratic equation.

The effects of a particular easing function can be visualised by animating the `y` coordinate of an object as time changes with a constant rate, such as bouncing it in and out.

<AnimationPlayer small name="tweening-visualiser" />

Because using timing functions to map a range of values is a really common pattern, it's possible to skip `map` entirely and pass the range to the timing function itself:

```ts
// This:
map(-300, 300, easeInOutCubic(value));
// Can be simplified to:
easeInOutCubic(value, -300, 300);
```

Interpolation functions

So far, we've only animated a single, numeric value. The [`map`](/api/core/tweening#map) function can be used to interpolate between two
numbers but to animate more complex types we'll need to use interpolation functions. Consider the following example:

```ts
// import { Color } from "@revideo/core";
yield *
  tween(2, value => {
    circle().fill(
      Color.lerp(
        new Color('#e6a700'),
        new Color('#e13238'),
        easeInOutCubic(value),
      ),
```

```
  );
 });
```

`Color.lerp` is a static function that interpolates between two colors:

<AnimationPlayer small name="tweening-color" />

:::tip

All [complex types](/api/core/types) in Motion Canvas provide a static method called `lerp` that interpolates between two instances of said type.

:::

Aside from the default linear interpolation, some types offer more advanced functions such as the [`Vector2.arcLerp`](/api/core/types/Vector2#arcLerp). It makes the object follow a curved path from point a to b:

```ts
yield *
 tween(2, value => {
  circle().position(
   // highlight-next-line
   Vector2.arcLerp(
    new Vector2(-300, 200),
    new Vector2(300, -200),
    easeInOutCubic(value),
   ),
  );
 });
```

<AnimationPlayer small name="tweening-vector" />

Tweening properties

The [`tween`](/api/core/tweening#tween) function is useful when we need to orchestrate complex animations. However, there's a better way of tweening individual properties. You may recall from the guide section that the following tween:

```ts
yield *
  tween(2, value => {
    circle().color(
      Color.lerp(
        new Color('#e6a700'),
        new Color('#e13238'),
        easeInOutCubic(value),
      ),
    );
  });
```

Can be written as:

```ts
yield * circle().color('#e13238', 2);
```

Here, we use a [`SignalTween`](/api/core/signals/SignalTween) signature that looks similar to a setter, except it accepts the transition duration as its second argument. Under the hood, this will also create a tween - one that starts with the current value and ends with the newly provided one.

We can chain multiple tweens together by calling the `to()` method on the returned object:

```ts
yield * circle().color('#e13238', 2).to('#e6a700', 2);
```

By default, property tweens use `easeInOutCubic` as the timing function. We can

override that by providing a third argument:

```ts
yield *
  circle().color(
    '#e13238',
    2,
    // highlight-next-line
    easeOutQuad,
  );
```

Similarly, we can pass a custom interpolation function as the fourth argument:

```ts
yield *
  circle().position(
    new Vector2(300, -200),
    2,
    easeInOutCubic,
    // highlight-next-line
    Vector2.arcLerp,
  );
```

`spring` function

The [`spring`](/api/core/tweening#spring) function allows us to interpolate between two values using Hooke's law. We need to provide it with the description of our spring and the `from` and `to` values. You can think of it as having a spring in resting position (the `to` value), stretching it all the way to the starting position (the `from` value), and then letting it go. The movement of the spring as it tries to reach the equilibrium is what we can use to dive our animations.

In the example below, we use springs to animate a position of a circle, but this method can be used for more things than just position.

<CodeBlock language="tsx">{springSource}</CodeBlock>

<AnimationPlayer small name="tweening-spring" />

Spring description

The first argument of the [`spring`](/api/core/tweening#spring) function expects an object that describes the physical properties of our spring. Motion Canvas ships with a few useful presets that you can use, such as `PlopSpring` and `SmoothSpring`. But it's possible to define your own spring:

```ts
const MySpring = {
  mass: 0.04,
  stiffness: 10.0,
  damping: 0.7,
  initialVelocity: 8.0,
};
```

- `mass` - Describes the inertia of the spring. How much force is required to accelerate and decelerate it.
- `stiffness` - The coefficient of the spring. Usually represented by `k` in Hooke's equation. It describes how stiff the spring is.
- `damping` - Over time, damping causes the spring to lose energy and eventually settle in equilibrium. You can set it to `0` to create a spring that oscillates indefinitely.
- `initialVelocity` - The initial velocity of the spring. You can set the `from` and `to` positions to the same value and give the spring some initial velocity to make it oscillate in place.

Settle tolerance

Notice how in our spring example, we provided the first spring with an additional value:

```ts
yield * spring(PlopSpring, -400, 400, 1 /*...*/);
//                          here ^
```

This optional argument is called `settleTolerance` and is used to define the minimal distance from the `to` value the spring should reach to be considered settled. The generator created by the [`spring`](/api/core/tweening#spring) function finishes **only** when the spring settles. By adjusting the tolerance we can make the animation finish faster, depending on our needs. In our example we animate the position so a tolerance of `1` means that the spring needs to be at most `1` pixel away from the `to` value.

Saving and restoring states

All nodes provide a [`save`](/api/2d/components/Node#save) method which allows
us to save a snapshot of the node's current state. We can then use the [`restore`](/api/2d/components/Node#restore) method at a later point in our animation to restore the node to the previously saved state.

```ts
// highlight-next-line
circle().save();
yield * circle().position(new Vector2(300, -200), 2);
// highlight-next-line
yield * circle().restore(1);
```

It is also possible to provide a custom [timing function](/api/core/tweening) to the [`restore`](/api/2d/components/Node#restore) method.

```ts
yield * circle().restore(1, linear);
```

Node states get stored on a stack. This makes it possible to save more than one

state by invoking the [`save`](/api/2d/components/Node#save) method multiple times. When calling [`restore`](/api/2d/components/Node#restore), the node will be restored to the most recently saved state by popping the top entry in the state stack. If there is no saved state, this method does nothing.

The example below shows a more complete example of how we can store and restore
multiple states across an animation.

<CodeBlock language="tsx">{saveRestoreSource}</CodeBlock>

<AnimationPlayer small name="tweening-save-restore" />

File Path: revideo-docs/platform/_category_.yml
label: Platform
position: 10
link:
  type: generated-index

File Path: revideo-docs/platform/introduction.mdx
---
sidebar_position: 1
slug: /platform/introduction

Introduction

The Revideo platform makes it easy to deploy Revideo projects in production. It works as follows:

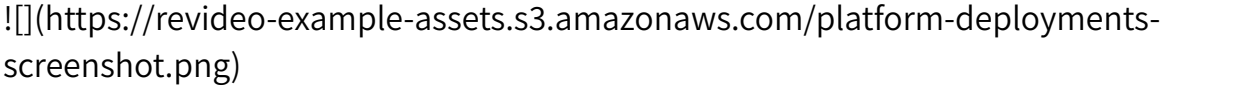
- You sign up to the Revideo platform using your Github account
- Within the platform, you can select a Github repository that contains a Revideo project of yours
- The platform will automatically set up a `/render` endpoint that handles fast parallelized rendering
- Every time you push to your Github repository, the platform will set up a new endpoint

The Revideo platform is not yet publicly launched, but is used by selected teams. If you want to get access to deploy a Revideo project of yours, you can sign up to our [waitlist](https://tally.so/r/mOz4GK) or directly email us at hello@re.video.

File Path: revideo-docs/platform/render-endpoint.mdx
---
sidebar_position: 2
slug: /platform/render-endpoint

Using your Render Endpoint

Your deployments are listed at
`https://re.video/platform/{org-name}/{repo-name}`. Here, you can find a list of deployments corresponding to every commit you made:



Rendering a Video

To render a video using one of your deployments, you can get its render url by clicking onto "Copy url". Now, you can send a POST request like this (see [here](/platform/render-endpoint#arguments) for all request parameters):

```bash
curl -X POST \
 https://api.re.video/v1/render/{your-deployment-id} \
 -H 'Content-Type: application/json' \
 -H 'Authorization: <your-api-key>' \
 -d '{
   "variables": {
     "text": "Hello world",
     "color": "#FF0000"
   },
   "settings": {
     "workers": 5
   }
 }'
```

Arguments

variables

any

The parameters of your video

callbackUrl?

string, optional

A callback to send the request response to. This is optional and probably not neccessary if you don't render very long videos - if not supplied, the request will remain open during rendering.

settings

An object with the following parameters

workers

number

The number of workers you want to parallelize the rendering job across

Return Value

The return value depends on whether a `callbackUrl` is provided in the request.

Without callbackUrl

If no `callbackUrl` is provided, the API will respond with a JSON object containing the render result. The response will have a status code of 200 if successful.

```json
```

```
{
  "resultUrl": "https://<revideo-storage-url>.com/<your-video-id>.mp4"
}
```

With callbackUrl

If a `callbackUrl` is provided:

1. The API will immediately respond with a status code of 200 and the body "ok".
2. Once the rendering is complete, a POST request will be sent to the provided `callbackUrl` with the following structure:

```json
{
  "resultUrl": "https://<revideo-storage-url>.com/<your-video-id>.mp4"
}
```

Error Responses

The API may return the following error responses:

- 401 Unauthorized: If the provided API key is invalid.
- 400 Bad Request: If the `deploymentId` is invalid, if the request body fails validation, or if the deployment is not running.
- 404 Not Found: If the specified deployment doesn't exist.
- 500 Internal Server Error: If there's an error getting the deployment, creating the render job, or during the rendering process.

Error responses will contain a plain text message describing the error.

File Path: revideo-docs/upgrading/_category_.yml
label: Upgrading
position: 9
link:
  type: generated-index

File Path: revideo-docs/upgrading/from_0.2.x_to_0.3.md
From 0.2.x to 0.3.x

The 0.3.0 release of Revideo moves the ffmpeg rendering process from a plugin into the core of the library. This means that it does now require a separate package to be loaded into the config inside `vite.confit.ts`. If you want to upgrade an existing project you need to do the following:

On 0.2.x versions, your `vite.config.ts` might look like this:

```ts
import {defineConfig} from 'vite';
import motionCanvas from '@revideo/vite-plugin';
import ffmpeg from '@revideo/ffmpeg';

export default defineConfig({
  plugins: [motionCanvas(), ffmpeg()],
});
```

On 0.3.x versions, you can safely remove the `ffmpeg` plugin:

```ts
import {defineConfig} from 'vite';
import motionCanvas from '@revideo/vite-plugin';

export default defineConfig({
  plugins: [motionCanvas()],
});
```

If you still run into errors, feel free to either open an issue on Github, contact us on Discord or using the chat in the bottom right corner of the website.

File Path: revideo-docs/upgrading/from_0.3.x_to_0.4.md
From 0.3.x to 0.4.x

The release of v0.4 comes with the following breaking changes:

renderVideo() accepts arguments as object and points to project file instead of vite config file

To make it easier to work with `renderVideo()` and `renderPartialVideo()`, the functions now accept arguments as attributes of a `RenderVideoProps` / `RenderPartialVideoProps` object.

Furthermore, they don't point to a `vite.config.ts` file anymore, but instead to your video's project file, which is `./src/project.ts` by default.

In Revideo 0.3.x, you might have called `renderVideo()` like this:

```tsx
const file = await renderVideo(
  'vite.config.ts',
  {fill: 'orange'},
  {logProgress: true},
);
```

To update to 0.4.x, wrap your arguments in curly braces, add the name of the argument as a key and replace `"vite.config.ts"`:

```tsx
const file = await renderVideo({
  projectFile: './src/project.ts',
  variables: {fill: 'orange'},
  settings: {logProgress: true},
});
```

Note that the following changes were made to the input arguments:

- in the `settings` object, "`name`", which refers to the name of the output file, was renamed to `outFile`. Also, you now have to append `.mp4` to `outFile`. In summary, if you previously had the argument `name: "myvideo"`, you should change it to `outFile: "myvideo.mp4"`.
- the `progressCallback` argument was moved inside of `settings`.

To check how you should structure your input arguments, you can take a look at the `renderVideo()` [docs](/api/renderer/renderVideo).

npx revideo serve serves the player through `localhost:4000/player` instead of `localhost:4000/player/project.js`.

In 0.3.x, when running `npx revideo serve`, you were previously able to obtain the player from `localhost:4000/player/project.js` in order to pass it to the `<Player/>`:

```tsx
<Player src="http://localhost:4000/player/project.js" controls={true} />
```

Now, in 0.4.x, you should just pass `localhost:4000/player` instead:

```tsx
<Player src="http://localhost:4000/player" controls={true} />
```

This change was made as we now start serving both the `project.js` file and the assets. If you have files in the `/public` folder of your Revideo project, you can now also use the assets from that project in another (NextJS) project that uses the `<Player/>`.